CrossTalk

April 2007 **The Journal of Defense Software Engineering** Vol. 20 No. 4

Agile Development

Report Documentation Page		*Form Approved OMB No. 0704-0188*

1. REPORT DATE **APR 2007**	2. REPORT TYPE	3. DATES COVERED **00-00-2007 to 00-00-2007**
4. TITLE AND SUBTITLE **CrossTalk: The Journal of Defense Software Engineering. Volume 20, Number 4, April 2007**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**32**	

ON THE COVER
Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

CrossTalk

Co-Sponsors:	
DoD-CIO	*The Honorable John Grimes*
NAVAIR	*Jeff Schwalb*
76 SMXG	*Kevin Stamey*
309 SMXG	*Randy Hill*
402 SMXG	*Diane Suchan*
DHS	*Joe Jarzombek*
Staff:	
Managing Director	*Brent Baxter*
Publisher	*Elizabeth Starrett*
Managing Editor	*Kase Johnstun*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); Ogden-ALC 309 SMXG; and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 25.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

"Lead, Follow, or Get Out of the Way"

Although made famous by Chrysler's Lee Iaccoca, the phrase was originally a quote from Thomas Paine. The quote strikes a chord with this month's theme of Agile Development. Businesses that just strive to keep up are at great risk of falling behind or, worse, becoming obsolete. On the other hand, businesses that are innovative and continually try to stay ahead tend to thrive. The businesses that are likely to succeed are those businesses who know what the customer wants before they even know they want it. Agile software and system development techniques are a perfect fit for such a business. Whereas traditional developers tend to be isolated from the customer, Agile methods require developers to be in tune with the needs of the customer. By understanding our customer's world, we can be innovative in meeting their needs. In Department of Defense (DoD) terms, an intimate relationship with our ultimate customer, the warfighter, helps us understand the capability needed to accomplish their mission. Their lives and our national security interest depend on us being in tune with their needs.

As developers and maintainers of DoD software, it's imperative that we are adequately agile to enable our warfighters to respond to continually changing threats and technologies. Getting new code to the field, however, involves much more than just developing the software; we must also address our policies and procedures for funding, testing, acquiring, training, and distributing software if we are going to be truly agile. Many emergency fixes are delivered at heroic speeds, but there is still progress to be made in order to intentionally deliver incremental capability real-time to need. It may be a far stretch from where we are today but imagine the possibilities of being able to tweak software in flight and receive instant feedback if it meets the user's need. A lot would have to change to make that leap, but I believe it is a worthy goal.

To address this challenge, I appreciate the opportunity to share continuing ideas to enhance Agile development. We begin with Dr. Alistair Cockburn's insights on the benefits of moving software incrementally and quickly through development in *What Engineering Has in Common With Manufacturing and Why It Matters*. Next, Esther Derby discusses some of the people skills that tend to be so critical in Agile development in *Collaboration Skills for Agile Teams*. We complete our theme articles with a contemplative look at Agile development from Dr. Richard Turner in *Toward Agile Systems Engineering Processes*.

In further discussions, my co-sponsors at the 309th Software Maintenance Group share one of their techniques for achieving Capability Maturity Model Integration Level 5 with *CMMI Level 5 and the Team Software Process* by David R. Webb, Dr. Gene Miluk, and Jim Van Buren. Consistent with Dr. Cockburn's assertion regarding the importance of decisions is Dr. David G. Ullman's discussion on making decisions in *"OO-OO-OO!" The Sound of a Broken OODA Loop*. We conclude with *Using Switched Fabrics and Data Distribution Service to Develop High Performance Distributed Data-Critical Systems* by Dr. Rajive Joshi.

We must find ways to lead – not follow. Our industry plays a critical role in providing warfighting capability that is unmatched anywhere in the world. As we consider Agile methods, we must realize that the DoD cannot afford to fall behind or become obsolete.

Kevin Stamey

Kevin Stamey
Oklahoma City Air Logistics Center, Co-Sponsor

What Engineering Has in Common With Manufacturing and Why It Matters

Dr. Alistair Cockburn
Humans and Technology

Software engineering is more like manufacturing than most people expect. Once we spot their similarities, we can apply the lessons learned over the last 50 years in manufacturing to software development. This article picks six lessons to apply to software development gleaned from the manufacturing industry.

It is generally considered frivolous to compare engineering – software engineering, in our case – with manufacturing. Manufacturing (so the reasoning goes) consists of making the same thing over and over, while software engineering is about making something different each time. In software engineering, coming up with the design and code is the hard part, while production is the easy part, sometimes as easy as publishing to the internet.

Software engineering is remarkably similar to manufacturing once we notice *decisions* as the product that moves through a network of people. In software development, people make decisions, hand those decisions to other people to build on, and (most importantly for this article) wait for other people to make their decisions. The *decision* in software development corresponds to a *part* in a manufacturing line: Both flow through a network, wait in queues at bottlenecks, have throughput delays, and so on.

With this equivalence in place, there is a very real parallel between design and manufacturing. This is useful to us because manufacturing has been studied heavily over the last 100 years, and we can learn from lessons in that industry.

In what follows, I shall focus on software development, but it should be clear that the same argument applies to every team-design activity, including engineering, theatre, publishing, and much of business.

Waiting for Decisions

We start by recognizing that in team-design activities, people wait on each other for decisions.

Figure 1 shows a simplified view of the dependencies between people in software development (it is missing the feedback loops, in particular). Figure 2 shows a more complete mapping of the decision dependencies, with some typical feedback loops. The feedback loops complicate matters, but do not change the basic results.

In Figure 1, the dependency of one person on another is shown with a large black arrow. The person at the tail of the arrow is making decisions and passing them to the person at the head of the arrow. A small pyramid represents the actual decision being passed from one person to another.

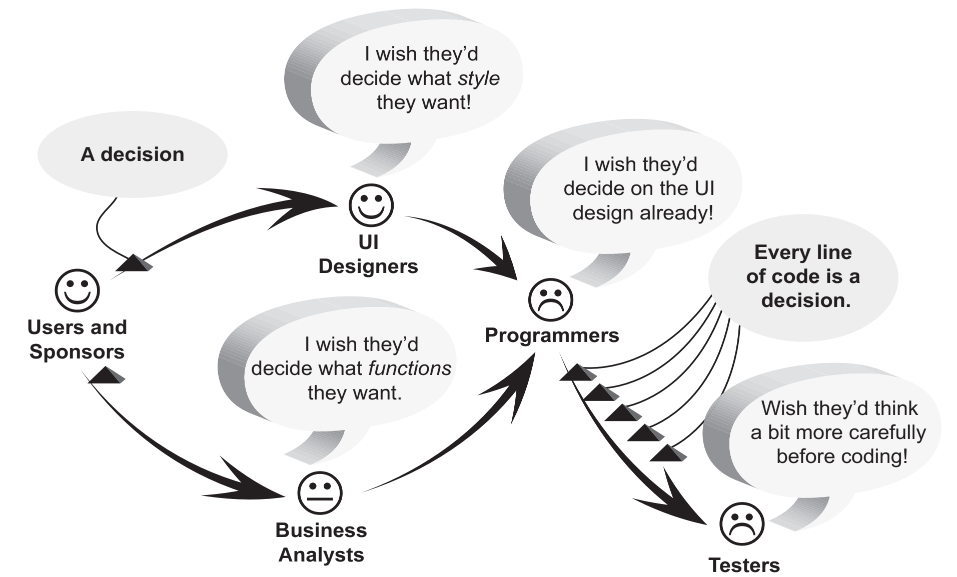
In Figure 1, we see the following:

- Business analysts and user interface (UI) designers waiting for users and sponsors to decide what functions and design styles they want.
- Programmers waiting for business analysts to work out the business rules and UI designers to allocate behavior to different pieces of the user interface.
- Testers waiting for programmers to finish their coding.

A nice thing about considering individual *decisions* as connecting people is that we can move away from stereotypes about how a company's process or decision-making activities *ought* to look, and instead focus on how it *actually* looks – what decisions actually get made by which people, and who is really waiting for whom.

There is no ideal software process any more than there is any ideal manufacturing process. Each company has its own strong-minded people who make a disproportionate number of decisions that might, in other companies, be made by people in other roles. Each company has its own shortage of UI designers, programmers, testers, or even sponsors, which causes its process to have a certain characteristic shape – people working overtime or sitting idle because other people can not get their work done fast enough. Each company has its own reasons to have a large, external test department, or perhaps no test department at all.

Figure 1: *People Wait on Other People for Decisions*


I wish they'd decide what *style* they want!
A decision
I wish they'd decide on the UI design already!
UI Designers
Every line of code is a decision.
Users and Sponsors
Programmers
I wish they'd decide what *functions* they want.
Wish they'd think a bit more carefully before coding!
Business Analysts
Testers

Different Bottlenecks, Different Processes

In any organization, we can find a backlog of decisions stacking up at some particular work group. This creates a *bottleneck*, which limits the speed of the overall team. Bottlenecks are of great concern in manufacturing and have received much study. The obvious thing to do is to increase the capacity of the bottleneck group – hire more people, or better people, or get better tools, and so on.

Sooner or later, however, the organization hits its limit as to what it can do to improve the speed of the bottleneck group. At that point, what comes into play is the process definition itself.

Figure 3 shows three different, but fairly typical organizations.

In the first organization, there are not enough UI and database designers to keep up with the work. We see decisions stacked up at their work centers. Assuming that this organization cannot or will not hire more UI and database designers, it should look at ways to have programmers and business analysts pick up sections of the UI designer's and database designer's work. Even assuming that UI design work is specialized, parts of that work can be automated, carried out by assistants, or handled by programmers.

In the second organization, there are not enough experienced programmers, and work requests stack up in front of them. In this second organization, the reverse is more the case. The programmers, being few and inexperienced, might need to have much of the problem *digested* for them as often as possible.

In such a situation, in which I participated, we recommended that the business analysts write quite detailed use cases (not containing the user interface, but containing the business rules more explicitly than we otherwise might), the layouts of the data needs, plus discussions of different business scenarios. The business analysts sat with the respective programmers as they started on each use case and discussed the use case, the scenarios, and the data. The business analysts left the paperwork with the programmers and made themselves available for discussions and tutorials as needed.

The process we came up with was aimed at minimizing the trouble the programmers had to undergo to understand the problem at hand. This is quite different from the process in the first organization.

In the third organization, the users and sponsors are notably missing from the discussion. What happens in these organizations is that the business analysts and UI designers end up making the business decisions and then sending those decisions (or running products) back to the users and sponsors for comment. The picture shows those requests for review stacking up in front of the users and sponsors.

The third picture also shows the programmers and database designers sending decisions back and forth to each other. Both groups need to come to agreement on the domain model and how that will be represented in the code and in the database.

In the third organization, the process might call for prototypes and early samples to be produced and put in front of the users and sponsors. Since those people have the least availability, the material should be as fully prepared as possible. Also, since close collaboration between the programmers and database designers is required, those teams should be seated together, or at least have frequent meetings and joint design reviews.

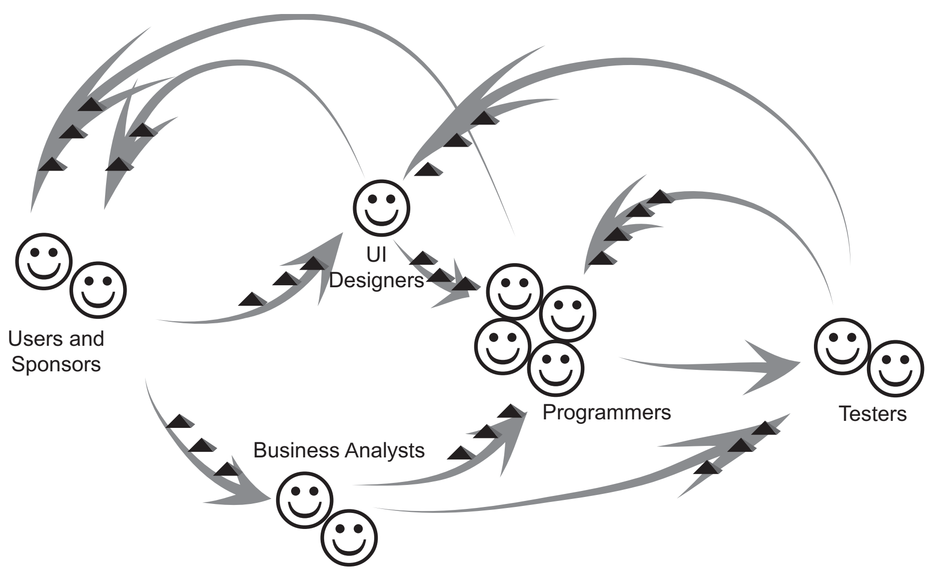


Figure 2: *Optimal Process and Strategies Vary With the Decision-Dependency Network*

There are the following two points to draw from these pictures:

- The organizations should be using different processes.
- These drawings help us to see how those processes should be different.

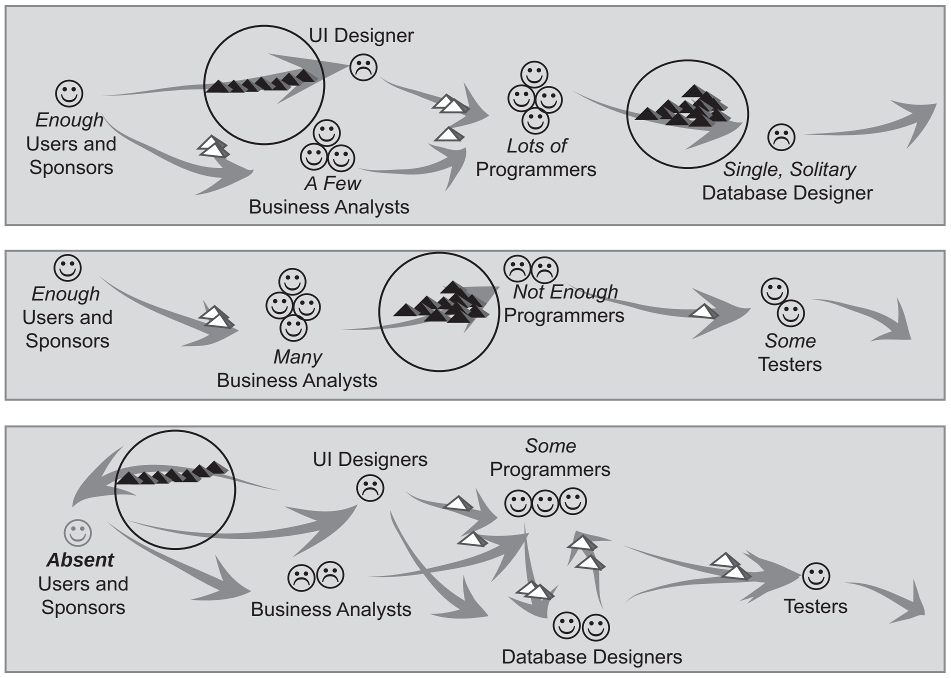
Lessons From Manufacturing

To apply the lessons from manufacturing, we need to recognize the life cycle of a decision:

- The decision gets made. It might be a business-level decision, a UI-design decision, or a decision about a particular line of code. The person making the decision does not really know at this point if it is a good decision or not.
- The decision gets reviewed internally. Part of reviewing a line of code is passing it through a test suite. Part of reviewing a UI design is putting it in front of a group of test users. Part of reviewing a business decision is putting it in front of sponsors and test markets. The decision fails the review, gets marked for adjustment, or passes.
- The decision gets pushed out into the world. At this point, the world makes a judgement about the quality of the decision and the decision makers get very useful feedback.

Even a very good decision has a finite lifetime, after which time it needs adjusting. A major goal of the development team is to get decisions reviewed, repaired, and sent

Figure 3: *A More Complete View of a Decision Dependency Network*



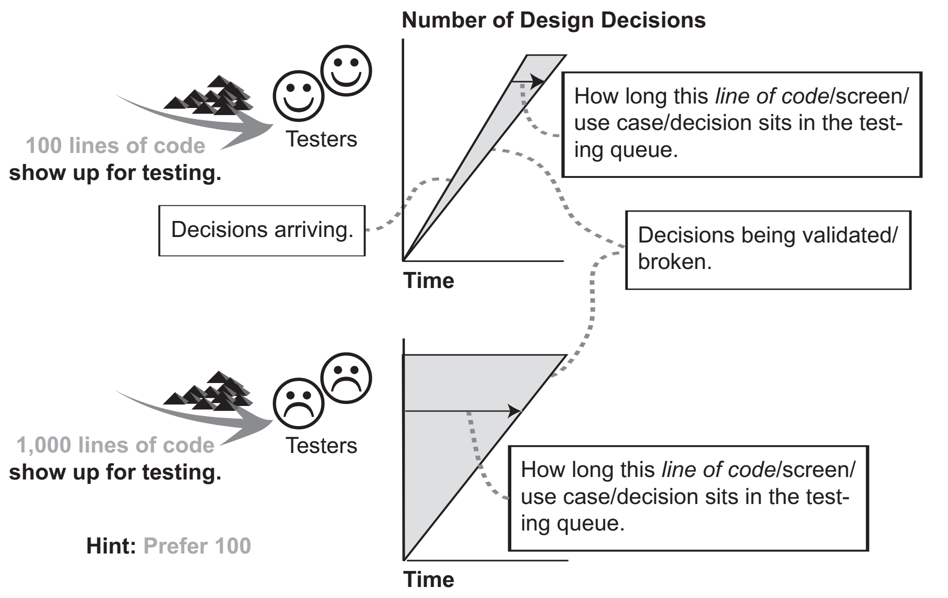

Number of Design Decisions
100 lines of code
show up for testing.
Testers
How long this line of code/screen/use case/decision sits in the testing queue.
Decisions arriving.
Decisions being validated/broken.
Time
1,000 lines of code
show up for testing.
Testers
How long this line of code/screen/use case/decision sits in the testing queue.
Hint: Prefer 100
Time

Figure 4: *Feed Systems Run More Smoothly With Small Transfer Sizes*

out into the world earning value as soon as possible. All the decisions waiting for internal and external review constitute internal inventory or work in progress (WIP).

Move Inventory Out

One of the lessons to draw from manufacturing is to reduce the WIP, that is, *get decisions out of development and into the business*. This is important in manufacturing, and it is also important in software development, because the value of decisions decays over time. Every moment a decision stays in the development cycle costs the organization money.

- Each *requirement* is a decision based on a business climate. When the business climate changes, the decision may become incorrect. If the software is not yet earning value for the company, the requirement is a waste.
- An *architecture* is a decision based on technology and business. If the technology changes before the software is earning value for the company, those decisions are a waste.
- Each line of code is a decision based on requirements, domain, technology, and aesthetics. If anything causes it to become obsolete before the software is earning value for the company, it is a waste.

To the extent that it is not earning value in the business, each decision loses value and quality with time. The more decisions stuck inside the pipeline, the more *decaying inventory* the organization is carrying.

Inventory stacks up quickly. Assume for reference an organization that is so fast that when a new requirement arrives, it can implement and deploy it by the next morning:

- The company with a one-day turnaround has about one day's worth of inventory lying around the office.
- The company with a two-week turnaround has about 10 days worth of inventory lying around the office.
- The company with a quarterly delivery system (assuming they deploy from fresh requirements every quarter) has about 100 days of inventory lying around.
- The company delivering a three-year project has 1,000 days of inventory (decaying) around them.

The message, in software as much as in manufacturing, is the following: *Get the inventory out the door and earning value!* Find ways to shorten and speed the pipeline.

Move Small Amounts, Continuously

The next lesson to draw from manufacturing is that, for the WIP (decisions still inside development), reduce the size of transfers between groups. Move small amounts often rather than stacking them up in large batches for long periods of time.

Figure 4 shows two ways of transferring work from the programmers to the testers.

In the first case, the programmers hand over 100 lines of code (each week, let's suppose). The testers get a regular weekly arrival rate of about 100 lines of code and have to integrate and test them against the rest of the system and against the known defect log.

The amount actually handed over will vary, of course, and the actual length of time needed to work through the new code will also vary. That variance is part of why small amounts should be handed over at any one time.

The lower part of Figure 4 shows the programmers handing about 1,000 lines of code to the testers (each quarter, that would be, to keep the rate of production about the same as in the upper picture).

The problem with the lower picture is that all 1,000 lines of code show up at one time. The responsiveness of the testing group suddenly becomes much more variable with the large arrival of an unknown number of bugs of varying sizes.

Equally bad is when the testers start handing bug reports back to the programmers and the programmers suddenly see a large spike of requests on their input queue coming from the testers (see the arrow in Figure 2, from the testers back to the programmers). The programmers are now juggling two work queues: requests for new features and requests for bug fixes.

Manufacturing groups have experienced and studied all the previous examples, and they concluded that these sorts of feed systems run best when *small* amounts of work get handed from one group to the next. The ultimate goal is to hand over just one part from one group or person to the next.

Toyota pioneered this idea in its lean or just-in-time manufacturing lines. They aim for *continuous flow*, the flow of just one piece of material from one person to another (what to do when a queue backs up is the subject of another lesson).

It is not clear exactly what *continuous flow* might mean in software development. Some design decisions affect large parts of the system, and some decisions can not be validated for a long time. However, the experiences in manufacturing are backed up by both mathematical models and experiences in agile software development.

It is rare to find development teams able to deploy fresh requirements every week, but I have been able to find a few teams who both deploy weekly *and* have a low enough defect rate that they get only a few requests a day. On one team I talked with, a person was assigned each day, on a rotation, to handle any incoming requests, whether bug reports or requests for small enhancements. That person would stop other work, do the work and redeploy the system before rejoining the main group. The average time to re-deployment was half a day. With such a small, steady flow of requests on the feedback queue, the team was able to keep from being diverted from their main assignment.

Cross-Train People

The literature on lean and agile manufacturing contains the recommendation to cross-train (training in multiple areas) people at adjacent stations. The idea is that when a small bubble of inventory shows up at someone's input, the neighboring person, having a spare moment, steps over and works it down. In this manner, small variances in work flow can be evened out and not disturb the organization's overall flow.

We see this in software development when programmers help testers, user interface designers, and business analysts, or when business analysts help testers. Unfortunately, programming is a technical enough activity that UI designers, testers, and business analysts are unlikely to be able to help the programmers. Programmers can help other programmers, though. We see in some companies that front-end developers, middle-ware developers, and back-end developers help each other when one of the groups has a sudden bump in work.

Extend the Network

All of these ideas are good – so good, in fact, that people using them soon find that their bottleneck lies somewhere in their supply chain, whether sponsors, subcontractors, or distributors. They start to draw the dependency network for the larger system in which they sit, and they start including their supply chain partners in their discussions.

Toyota is well known for working with its suppliers. Less well known are cases of software development groups doing it. The same team I referred to earlier, using the daily programmer rotation for fixes and enhancements, also wrote automated acceptance tests for their subcontractor's part of the system. They reasoned that their time was better spent writing automated acceptance tests and catching bugs on arrival than debugging and finding those same faults in the integrated system when their supplier's code broke. The supplier was, of course, surprised and delighted to find they did not have to write the automated acceptance tests.

The lesson from Toyota and the other companies who are streamlining the wider network is the following: *The wider the network of* we*, the faster we all go.*

Who's Writing About This?

Once we see the mapping between manufacturing and team design activities, suddenly a lot of literature becomes available.

Toyota's production system (also called The Toyota Way and lean manufacturing) is widely documented. A good place to start is with *The Toyota Way Fieldbook* [1].

The application of lean manufacturing principles to design work is described in *Managing the Design Factory* [2], *Product Development for the Lean Enterprise* [3], and *The Elegant Solution* [4].

Tom and Mary Poppendieck describe in several books [5, 6] how lean manufacturing principles fit software development. *Agile Software Development: The Cooperative Game* [7] contains an experience report from a software product company (Tomax) that includes its customers in its dependency network.

Elihu Goldratt wrote about bottleneck stations in manufacturing [8] and then widened the discussion to constraints in general (*theory of constraints*) [9]. David Anderson applied the theory of constraints and queue size to software projects [10]. I have written about strategies for dealing with bottlenecks that have reached their capacities [11].

Summary

It is not immediately obvious that software development teams can learn from manufacturing. However, once we chart the network of dependencies between people in a software development organization and make the shift to think of *decisions* as comprising the team's *inventory*, then the parallels become startlingly clear.

We learn six lessons from the parallels:

- Drawing the decision-dependency network helps us spot the *bottleneck* stations, where decisions-to-be-made are piling up.
- From the different decision-dependency networks in various organizations, and their varying bottlenecks, we can see how the *optimal process* varies from organization to organization.
- *Move inventory out.* Decisions decay over time, so it is important to find ways to shorten the pipeline from arrival of a request or decision to the deployment of the system.
- *Move small amounts, continuously.* Transferring large amounts of inventory (decisions, in our case) between workers causes unpredictable variations in the organization's output. It is better to move small numbers of decisions more often. This reinforces the idea of incremental development, with the smallest increment size possible.
- *Cross-train people.* When people can help each other across specialties, they can move quickly to eliminate small bubbles in each others' input queue, thus smoothing the organization's output.
- *Extend the network.* By widening the network included in the dependency analysis and queue-size reduction, a company can smooth its own input stream and simplify its work.◆

References

1. Liker, J., and D. Meier. The Toyota Way Fieldbook. McGraw-Hill, 2005.
2. Reinertsen, D. Managing the Design Factory. Free Press, 1997.
3. Kennedy, M. Product Development for the Lean Enterprise. Oaklea Press, 2003.
4. May, M. The Elegant Solution. Free Press, 2006.
5. Poppendieck, M., and T. Poppendieck. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley, 2006.
6. Poppendieck, M. and T. Poppendieck. "Lean Software Development." C++ Magazine (Fall 2003).
7. Cockburn, A., Agile Software Development: The Cooperative Game. 2nd Edition. Addison-Wesley, 2006.
8. Goldratt, E. The Goal. North River Press, 1992.
9. Goldratt, E. Theory of Constraints. North River Press, 1999.
10. Anderson, D. "Managing Lean Software Development With Cumulative Flow Diagrams." Proc. of the Borland Conference, 11-15 Sept. 2004, San Jose, CA.
11. Cockburn, A., "Two Case Studies Motivating Efficiency as a Spendable Quantity." Humans and Technology Technical Report HaT TR 2005.00 <http://alistair.cockburn.us/index.php/Two_Case_Studies_Motivating_Efficiency_as_a_%22Spendable%22_Quantity>.

About the Author

Alistair Cockburn, Ph.D., is an expert on object-oriented (OO) design, software development methodologies, use cases, and project management. He is the author of *Agile Software Development*, *Writing Effective Use Cases*, and *Surviving OO Projects* and was one of the authors of the Agile Development Manifesto. Cockburn defined an early agile methodology for the IBM Consulting Group, served as special advisor to the Central Bank of Norway, and has worked for companies in several countries. Many of his materials are available online at <http://alistair.cockburn.us>.

1814 East Fort Douglas CIR
Salt Lake City, UT 84103
Phone: (801) 582-3162
Fax: (775) 416-6457
E-mail: acockburn@aol.com

Collaboration Skills for Agile Teams©

Esther Derby
Esther Derby Associates, Inc.

Beyond technical skills, Agile Development depends on effective interactions and collaboration. In this article, Esther Derby outlines key collaboration skills that help teams maintain productive relationships, avoid destructive conflicts, and benefit from everyone's best ideas.

Agile Development requires close collaboration. But most programmers and testers have been trained to value competition and individual effort through their schooling and professional experiences.

Is it any surprise that working collaboratively on an agile team may not come naturally? Along with learning new technical skills and development methods, successful agile teams learn – or strengthen – interpersonal skills. Teams that do not invest in these skills may see improvement but miss the potential for high-performance.

In my work, I see three areas that help boost a team to the next level of performance. They are the ability to do the following:

- Give congruent feedback.
- Navigate conflict.
- Think and decide together.

In this article, I outline each of these areas and talk about pitfalls for teams that lack these essential skills.

Give Congruent Feedback

In more traditional organizations, the manager or project manager makes assignments and follows up to make sure the work is on track. People retreat to their own cubicles and may communicate via instant messaging or e-mail, even when the other person is only down the hall.

On agile teams, the team organizes its own work, making commitments to all on the team. Ideally, team members are in the same open workspace, and agile methods emphasize frequent interaction and face-to-face communication. This increases the probability that sooner or later, one person's behavior will irritate someone or someone will fail to meet a commitment made to a peer.

When team members cannot talk to each other about missed commitments or behavior that affects the working relationship, resentment builds up. However, taking problems to a coach or manager creates an unhealthy triangulation – like the tattletale on the playground.

Further, there is a cost to withholding feedback. Not long ago, a developer approached me for advice about a problem team member. The developer reported that one team member was alienating other team members. No one wanted to work with him, and most of the team refused to pair program with him.

As the story unfolded, I learned that the offending team member, Joe, had an unpleasant habit: He picked his nose. The team coach had made vague references to good manners in a team meeting, but the problem persisted (not surprising, since general pronouncements are not a substitute for clear, direct feedback).

"Conflict is normal and inevitable when more than one person is on a project. That is not necessarily bad; lack of conflict indicates apathy, not harmony."

By the time I talked to the developer, the problem had been going on for three months. Joe was confused by the way people were treating him. The team was losing the benefit of his knowledge, and it was showing up in the quality of the code.

Joe's habit was a problem. The bigger problem was that no one on the team knew how to talk to him about it.

The following is a simple feedback model to help team members have a feedback conversation:

- Create an opening to give feedback.
- Describe the behavior or result without using labels.
- State the impact (on you, the feedback giver, or on the team).
- If necessary, make a request.

This formula helps people stick to *I* language and avoid labels and blame. People are more likely to make a change when the feedback giver does not blame, shame, or evaluate the feedback receiver. Feedback is information, and the over-arching goal of feedback is to improve work and social relationships.

With some coaching, the developer approached Joe directly. He worked up his courage and told Joe about his habit and the effect it had on him. The developer was surprised to learn that Joe was completely unaware of his habit. Joe was embarrassed, but also grateful that someone had finally told him.

All teams have disappointments and friction. Contrary to a widespread fear, congruent feedback does not damage relationships; it increases trust and openness. Clear and early feedback keeps small irritations from growing into major resentments [1].

Navigate Conflict

Conflict is normal and inevitable when more than one person is on a project. That is not necessarily bad; lack of conflict indicates apathy, not harmony [2]. The way people handle conflict determines whether a conflict is productive or destructive. People whose work is interdependent are more productive when they learn to recognize the causes of disagreements and navigate conflicts productively [3].

In my work with groups, I see four basic sources of interpersonal conflict: misunderstanding, focusing on positions, differing values, and bringing up past history.

Misunderstanding

Sometimes people disagree because they do not understand each other. Sometimes the misunderstanding is over the use of a term that has many meanings (*system testing* is a common culprit; *done* is another). Or, people may not understand the details under discussion.

I attended a planning meeting where the participants argued in circles for 20 minutes about which of three approaches to follow for a release. I felt confused as I tried to follow the discussion.

"Wait a minute," I said. "Can someone write down the different options you're considering?"

By the time the team members finished writing down the options, it was clear there were actually four main options – and three variations.

The simplest strategy when people disagree is to review the data and write it down where everyone can see it.

Focusing on Position

Many of us grow up with the idea that one side wins and one side loses. That leads us to focusing on a *position* – pushing our favored solution [4] rather than talking about the problem and how we might solve it in a mutually agreeable way.

To bring focus back to the problem, ask *what problem are we trying to solve?* Then ask about the concerns behind both (or all) positions. When team members see the interests behind the position, they may find common ground or see a third option that incorporates interests from both sides.

A variation on this type of conflict comes from considering too few options. One group I worked with fought over decisions every week. In each case, they looked at only two options: either we do A or we do B. Having only two options is inherently polarizing. Generating additional options reduces unproductive conflict and increases analytical thinking.

Differing Values

When people are unable to reach agreement, even when both options would solve the problem and both parties seem interested in moving forward, they may be at odds over core beliefs about what is *true* and *good*.

Surface the values behind an option by asking about the strengths of the option. The words that people use to describe the strengths offer a clue about what the person values. Look for a third (or fourth or fifth) option that includes the top strengths from each option.

For most teams, the majority of the disagreements they face fall into the previous three categories: misunderstanding, focusing on position, or differing values. When team members learn to recognize the source of the disagreement, they can move quickly to resolve the disagreement – without being disagreeable.

Past History

When people are not able to give congruent feedback and navigate disagreements productively, simple disagreements escalate into ruptured relationships which show up as cheap shots and sniping. Trying to resolve the argument on the merits of the facts will not work because the argument is not about the facts. When the disagreement reaches this point, it is about the belief that the warring parties hold about each others motivations and intentions [5].

Ruptured relationships are poison on any team. On an agile team, where achieving the goal depends on every team member's contribution, ruptures can be fatal. Unless at least one person is willing to improve the situation and look at how he or she has contributed, there is little hope of positive resolution. The good news is that when people learn how to give congruent feedback and know how to recognize sources of disagreement, working relationships are not likely to sink to that level.

"The simplest strategy when people disagree is to review the data and write it down where everyone can see it."

Knowing the sources of conflict does not ensure people navigate conflict successfully. Most people have a default approach to conflict, which may or may not be effective depending on the situation. There are five basic approaches to conflict.

1. **Competition** assumes that one person will win and the other will lose. People press their own preferred solution rather than seek to understand the other person's interests. People who approach conflicts as competition may argue their point and undermine the other's point.
2. In **collaborative problem solving**, both parties seek to find options that will satisfy both of them.
3. When one person gives into another's wishes without representing his or her own interests, it is called **yielding**.
4. Sometimes people do everything they can to **avoid** a conflict. They pretend the difference does not exist to save themselves from the unpleasantness of confrontation.
5. In **compromise**, people try to meet halfway. Each gives up some of what he wants and achieves some of what he wants. Compromise is common, though not always satisfying since no one is completely happy with the solution.

All of these are valid and useful ways to approach conflict in some situations. And each can be destructive when misapplied. Members of successful teams have the self-awareness to recognize their own preferred styles and know when to move out of their default approach to conflict.

Competition can damage relationships, especially when every disagreement or conflict becomes an *I win/You lose* proposition. Competition over small issues feels like browbeating or bullying. When one or more team members over-rely on this conflict approach, relationships and productivity suffer.

Collaborative problem-solving might not be helpful when there is a clear downside to meeting the other's interest, for example, if the other person wants to pursue an illegal or unethical action. A collaborative approach also takes time in order to uncover interests, generate options, and reach a mutually satisfying outcome. It is worth the time when long-term relationships are at stake, but may not be when time is of the essence or the relationship is transitory.

Yielding is fine when one person does not have much investment in the outcome and the other person does. Yielding hurts when it is habitual – one person always gives in to the other. Others may perceive habitual yielders as doormats and walk all over them. Habitual yielding carries a cost. For example, a team that always says yes to the customer's requests during iteration planning meetings avoids the short term stress of an unpleasant conversation. But in the long term, the team risks burnout if they struggle to deliver on unrealistic commitments. They risk their reputation as trustworthy professionals if they fail to deliver. Over time, habitual yielding results in resentment, depression, anger, and contempt [6].

Avoidance may be a reasonable course when there is nothing to gain by pursuing an argument; savvy team members learn how to pick their battles.

Compromise often ends in a half-horse, half-camel solution that is not fully satisfying to anyone, and can cause teams to miss novel solutions. But compromise is the best option when it is clear that a collaborative solution is not feasible.

Most people have a preferred style for approaching conflict. Teams suffer when people on the team approach every conflict with the same style, regardless of what is at stake and without consideration for maintaining important relationships.

Think and Decide Together

On many traditional teams, the manager makes important decisions. But agile teams work best when they have the authority to make decisions that affect their own work (within the context of organizational standards). In order to make timely decisions that the team can support, teams need three broad skills:

1. Generating ideas.
2. Narrowing the number of options.
3. Reaching agreement [7].

When one or more of these elements is missing, teams struggle to make decisions. The good news is that most agile teams can learn techniques that will help them self-facilitate without investing in extensive facilitation training.

Generating Ideas

A combination of individual brainstorming and affinity clustering can help a team generate many ideas in a short period of time [8]. Pairing these two techniques allows the group to integrate ideas and find common threads.

Narrowing the Number of Options

When I see a team stuck evaluating alternatives, it is usually for one of the two following reasons: 1) People do not have a common definition of the options under discussion (a common source of disagreement described earlier), or 2) the group is talking about all the options at the same time.

Overcoming the second problem takes some discipline: Evaluate each option on its own before comparing options to each other.

Draw two lines on a piece of flip-chart paper, creating three columns, as shown in Table 1. List the pros and cons of the options in the first two columns. Make a note of what is interesting about the option in the third column. Answer all three questions for one alternative before moving on to the next. After the group has completed this activity for all the options, it is usually obvious that some of the ideas are unsuitable.

Table 1: *Pros and Cons*

Alternative 1		
Pros	Cons	Interesting

Reaching Agreement

Teams need a way to test their agreement and discuss concerns before they arrive at a final agreement. A simple hand sign can help a team gauge their level of agreement:

- Thumbs up = I support this proposal.
- Thumbs sideways = I'll go along with the will of the group.
- Thumbs down = I do not support this proposal and wish to speak.

If all thumbs are down, eliminate the option. On a mixed vote, listen to what the thumbs-down people have to say, and re-check the agreement. Thumb-sideways helps show where support is lukewarm.

Finally, teams need to decide how they will decide and identify a fall-back decision rule (in case they are unable to reach agreement).

Conclusion

With skills in these areas – congruent feedback, navigating conflict, and thinking and deciding together – teams have a basis to work through the inevitable friction. Without collaboration skills, teams struggle to manage both the upside and downside of collaboration. In my work, I see a predictable progression for teams adopting agile methods.

In the first months, teams concentrate on structures: daily stand-up meetings, iteration planning meetings, and mechanisms to keep progress visible.

Next, they face the difficulties of organizing their working in short (one week to 30 days) iterations.

When those pieces are in place, teams typically recognize that their engineering practices are not adequate to the job and attack those.

Finally, teams realize that in order to work effectively with their customer and with each other, they need collaboration skills. As Jerry Weinberg famously said, "It's always a people problem."

However, pushing collaboration skills before a team recognizes the need is not helpful. Adults are motivated to learn when they see the value of new ideas for solving the problems they face. When agile teams recognize that collaboration skills will help them deliver valuable software, perhaps with some nudging from their coach, they are eager to learn.◆

References

1. Seashore, Charles N., Edith Whitfield Seashore, and Gerald M. Weinberg. What Did You Say? The Art of Giving and Receiving Feedback. Bingham House Books, 1992.
2. Eisenhardt, Kahwajy, and L.J. Bourgeois. "How Management Teams Can Have A Good Fight." Harvard Business Review (July/Aug. 1997).
3. Katzenbach, Jon R., and Douglas K. Smith. The Wisdom of Teams: Creating the High-Performance Organization. Harper Collins, 1999.
4. Fisher, Roger, and William Ury. Getting to Yes: Negotiating Agreement Without Giving In. Penguin Books, 1983.
5. Stone, Douglas, Bruce Patton, and Sheila Heen. Difficult Conversations: How to Discuss What Matters Most. Penguin Books, 1999.
6. Satir, Virginia, John Banman, Jane Gerber, and Maria Gomori. The Satir Model: Family Therapy and Beyond. Science and Behavior Books, 1991.
7. Kaner, Sam. Facilitator's Guide to Participatory Decision-Making. New Society Publishers, 1996.
8. Sanfield, Brian R. The Workshop Book: From Individual Creativity to Group Action. New Society Publishers, 2002.

About the Author

Esther Derby is known for her work in helping teams grow to new levels of productivity and coaching technical people who are making the transition to management. She is one of the founders of the Amplifying Your Effectiveness Conference and is co-author of *Behind Closed Doors: Secrets of Great Management.* Her latest book is *Agile Retrospectives: Making Good Teams Great.* Derby has a master's degree in organizational leadership and more than two decades experience in the wonderful world of software.

3620 11th AVE S
Minneapolis, MN 55407
Phone: (612) 724-8114
Fax: (612) 724-8115
E-mail: derby@estherderby.com

Toward Agile Systems Engineering Processes

Dr. Richard Turner
Systems and Software Consortium

Agile software development approaches have been highly successful in a variety of domains. Could they be effective if applied to systems engineering? This article begins a discussion to answer this question by comparing core agile characteristics to those of traditional systems engineering.

The concept of agility is cropping up more and more often throughout the defense and commercial development worlds. It has found its way into the Quadrennial Defense Review, acquisition plans and procurement requests, and even into the language of defense executives[1]. Promises of faster deployment and evolutionary capability, delighted customers and users, and fewer late-occurring acquisition problems are irresistible to the resource-strapped, schedule-limited, and continuously harried program managers and acquisition executives.

However, where can the agile benefit really accrue? Primarily associated with software development, does the concept play into the large systems development that is typical of the defense environment? How does agility apply to the critical systems engineering processes? While research is needed to fully answer these questions, we can begin to identify touch points that on the surface seem ripe for agile approaches.

This article presents some thoughts on agility and systems engineering – how systems engineering can be more agile and how it can support agility in other disciplines. It is a concept discussion, not a specific how-to article. However, looking at systems engineering through the agile lens can extend the dialogue that began between agile and plan-driven software proponents into the systems engineering world [1].

First of all, why should we care about agility within systems engineering? Table 1 identifies some of the changes in the environment facing systems and software developers. The rapid change in threats, requirements, and programmatic parameters has pushed traditional approaches to the limits of their capabilities. As a result, there is a growing zeitgeist that somewhat unfairly casts traditional systems engineering as a holdover from the 1950's and 1960's and as a part of the systems acquisition and development problem. Agilists generally view systems engineering as rigid and waterfall-based, overly process-bounded (MIL-STD-499, MIL-STD-1521, Institute for Electrical and Electronics Engineers [IEEE]-15288). Myopically focused on early correctness, systems engineering can seem to value precision over accuracy and completeness over rapid user satisfaction. Figure 1 shows the traditional systems engineering *V-model* as it was developed for large systems. The model has evolved over time, but the fundamentals still provide a basis for the life cycle used by defense system acquirers. That is, establish requirements, establish an architecture, decompose the system into subsystems, design the subsystems, build the subsystems, test the subsystems, integrate the subsystems, and then test the system.

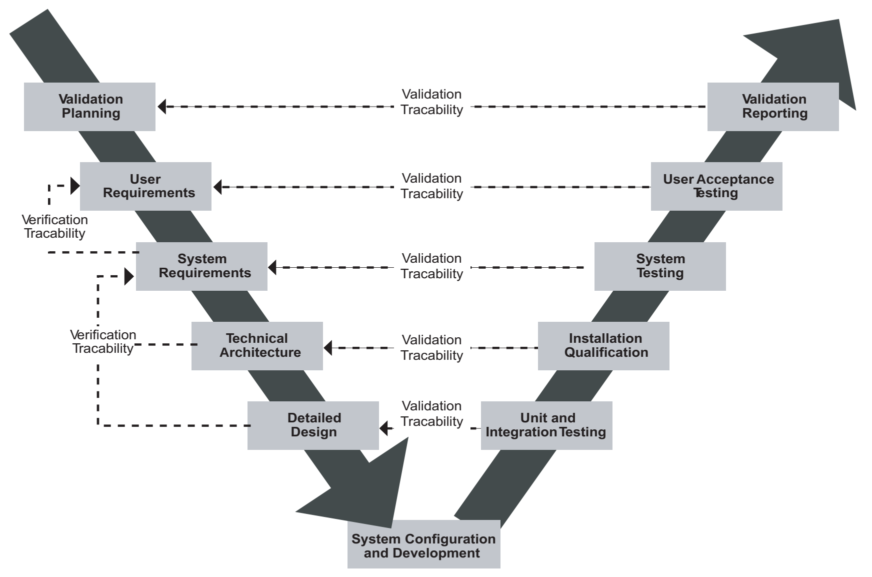
At the same time, agile approaches are portrayed as the promised land. Praised as a panacea for all the developmental ills, agile approaches claim victory over rapid change, increased complexity, emerging requirements, and the ubiquitous schedule-busting integration fiascos. Figure 2 (see page 12) shows a typical agile process. Note the iterative rather than sequential nature. While an iteration could represent a mini-waterfall, that is not always the case, particularly in risk reduction activities.

Of course, neither of the broad characterizations of the approaches is particu-

Table 1: *Some Software-Intensive System Trends*

Traditional Development	Current/Future Trends
• Standalone systems	• Everything connected (maybe)
• Relatively stable requirements	• Rapid requirements change
• Requirements determine capabilities	• Commercial off-the-shelf (COTS) capabilites determine requirements
• Control over evolution of custom systems	• No control over evolution of COTS products
• Enough time to keep stable	• Ever-decreasing cycle times
• Stable jobs	• Outsourced jobs
• Failures locally critical	• Failures broadly critical
• Completely defined systems with specific functionality	• Complex, adaptive, emergent systems of systems
• Repeatability-oriented process, maturity models	• Adaptive process models

Figure 1: *V-Model of a Conventional, Large-System Development Process*



Agility and Process Maturity

It is important to understand that agility is not anti-process, but can conform to Capability Maturity Model Integration (CMMI®) and other process standards. In fact, the Systems and Software Consortium is currently developing a Process Implementation Indicator Description table for CMMI Lead Appraisers to use in appraising agile projects.

Agile concepts in many ways embody Level 5-ness by continuously improving or adjusting processes. By conducting a retrospective/reflective activity after each iteration, recommendations for improvement can be immediately implemented. Agile measures can then confirm or contradict the value of changes within the next few iterations rather than waiting for the next project.

Agile does not specifically address the organizational aspects of many process standards (e.g. Organizational Process Focus, Organizational Process Definition, and so forth in CMMI), but is not a stumbling block to satisfying them. Usually, there needs to be agile instantiations in the set of organizational standard processes to limit tailoring confusion and support agile approaches.

larly accurate as stated, but they do provide insight into the turmoil that has continued to bubble. Regardless of the hype, there is no denying the need for leaner, more responsive development processes. If agile approaches can be harnessed in systems as well as software engineering, they are certainly well worth the effort.

But what, you ask, is *Agile*? There are nearly as many definitions of Agile as there are Agile practitioners. I believe, however, that there are common, key aspects that must be present to capture the essence of Agile. Table 2 captures my own essential list of agile features.

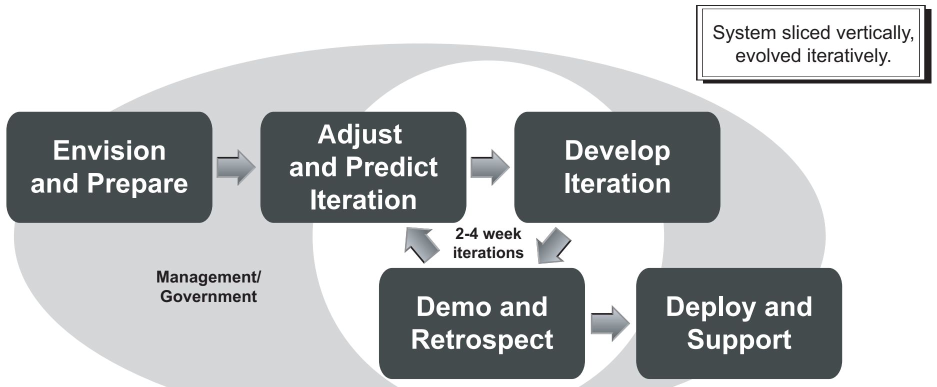
Agility and Systems Engineering Processes

So how do these attributes apply to systems engineering? How can we mature systems engineering to encompass these attributes? Let us look more closely at a few of the attributes that seem to address the engineering process.

Systems Engineering as a Learning-Based Process

One of the characteristics of traditional project management, and by implication much of traditional systems engineering, is the assumption by all stakeholders that foreknowledge is perfect. We can define complete, consistent, testable, and buildable requirements; decompose perfect requirements to perfect specifications; accurately estimate effort, cost, and schedule for the specifications; schedule work according to this information early in the program; and measure progress using earned-value management or similar techniques. While program managers, executives, sponsors and fund providers may believe this, engineers know that with any sufficiently complex system, particularly unprecedented systems, it is unrealistic to assume this kind of knowledge. As Philip Armour said, … *for the most part, engineers do not know how to build the systems they are trying to build; it is their job to find out how to build such systems* [3]. That is why systems engineering can be visualized as a set of tools and approaches that allow us to seek information that fills the gaps in the initial descriptions. By doing so, it adjusts the development to fit the reality of what we have learned. Trade studies, requirements analysis, demonstrations, prototypes, models, design evaluations, allocation analyses, and verification and validations are all ways to learn about the system being developed. So, there is no fundamental reason systems engineering cannot be considered a learning process. Unfortunately, the traditional view of the systems engineering V-model often is interpreted so that it provides only a limited, *one-time through* chance to learn. By reinterpreting the V-model from an agile perspective and using timely iterative feedback, the learning process can be richer.

Figure 2: *Disciplined Agility Process, Basic Model* [2]



Systems Engineers (SEs) as Focused on Customer Value

SEs are often isolated from the customers because their customers are considered fully represented by the predefined requirements and operational concepts. These ostensibly perfect requirements are generally value-neutral, with no sense given to their importance in relationship to each other, save some very high-level key performance parameters or possibly some value thresholds within a particular requirement. This puts the learning systems engineer at a huge disadvantage by debilitating an entire dimension of the trade space: the ability to consider the relative value to the customer of a requirement in deciding to defer or relax it in order to meet some other requirement or for other engineering reasons. The tradeoff between cross-cutting aspects like safety, security, maintainability, and performance has been identified as the number one risk by a University of Southern California survey of systems and software engineers [4]. The relative importance of the requirements must be interpolated using the engineer's experience, physical constraints, and domain knowledge so that fundamental engineering decisions can be made. It would be much easier if the requirements were not only clear and concise but also ranked in terms of importance. There is nothing to prevent including this dimension by having more complete and multi-faceted interfaces with the customer, but the traditional systems engineering requirements activities generally do not support it.

Systems Engineering With Short Iterations

Because systems engineering has been often viewed as a one-pass process (the strict V-model), iterations of systems engineering may sound foreign. However, there are ways to do iterative systems engineering. Prototyping, model-

® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ing, demonstrating, and testing can all be iterative within an integrated systems engineering and development cycle. The difference in truly agile iterations is that each of these should describe a complete operable system with functionality that is valuable to the customer. However, in the early systems engineering phases, deployable operational aspects may not be as valuable to the customer as reduced risk, requirements validation, operations concept validation, interface and interoperability verification, or technical feasibility. Systems engineering activities in later iterations are focused on operational capabilities. Development processes where systems engineering is seen as an up-front process and the SEs complete their trades and decomposition tasks and then move on to another program until needed for validation (sometimes referred to as the *do it once and the SEs do lunch* approach) are not conducive to iterative work. One of the most creative ways of envisioning systems engineering iterations is Barry Boehm's characterization of systems engineering as a Command and Control, Intelligence, Surveillance and Reconnaissance (C2ISR) activity (Figure 3), consisting of numerous Observe, Orient, Decide, Act (OODA) loops and ongoing intelligence, surveillance and reconnaissance tasks [5]. This counters the traditional cycle of requirements, delay, and surprise.

Attribute	Comment
Learning attitude	• Take advantage of lessons learned and adapt both processes and systems to meet customer needs.
Focus on value to customer	• Customer prioritizes requirements and progress is measured by operational features.
Short iterations delivering value	• Goal of each release is a working system. • Rolling planning horizon. • Risk-driven, reality-based iteration planning.
Neutrality to change (design processes and system for change)	• Change is seen as inevitable; ergo *embrace change* applies.
Continuous integration	• Integration is an ongoing activity. • Integration and testing are as automated as possible.
Test-driven (demonstrable progress)	• Tests are written before any other artifacts (design, code). • Capabilities (requirements) are defined by the tests (empirical evidence) that validate them.
Lean attitude (remove no-value-added activities)	• As little ceremony as necessary; just enough (or just too little) process. • Decisions delayed until latest *feasible* time.
Team ownership	• Team has primary responsibility and authority over its own plans and processes. • Quality/performance is everyone's responsibility.

Table 2: *Key Characteristics of Agile*

Systems Engineering and Neutrality to Change

This involves the architectural and design approach more than pure systems engineering. Unless systems engineering performs its activities and processes with an eye toward supporting change rather than avoiding or denying it, change will become an enemy (rather than an annoying but faithful family member). System engineering can use change as a dimension in its trade studies, evaluate the ease of modification or extension within architectural reviews, and even add requirements and design constraints that support change neutrality.

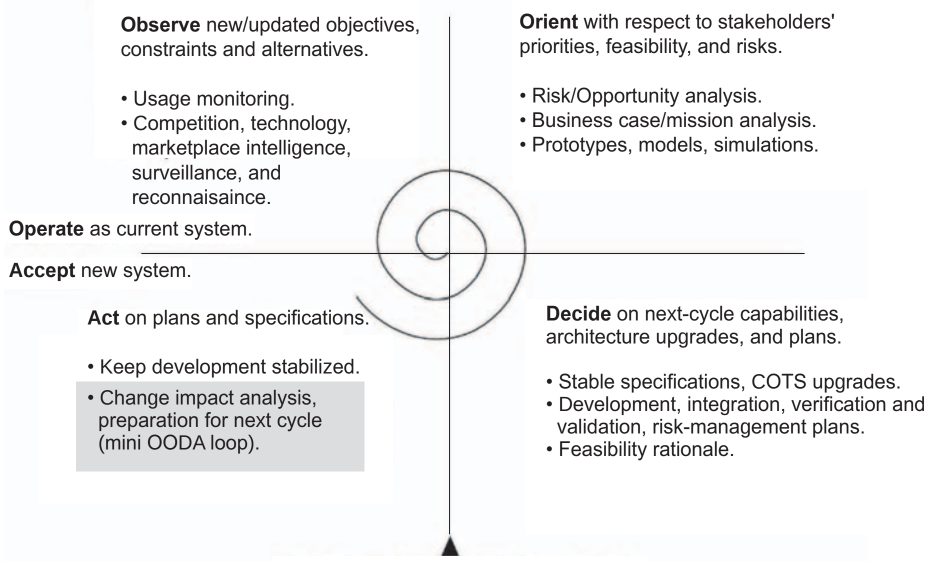
Systems Engineering, Continuous Integration, and Test Driven Development (TDD)

Once we accept the idea that SE iterations are feasible, then continuous integration and TDD are not as problematic. In order to provide an operable system that demonstrates value, there must be ways to maintain the configuration over time and use it as initial validation of operational capability, interoperability, and interface quality. Most likely done in a completely simulated or hardware-in-the-loop environment, frequent integration and requirements based testing (especially where there are external components that you may or may not control), can identify anomalies, misinterpretations, and downright errors in the interface specifications or implementations much earlier than traditional late-in-the-process integration. This does require a change in the once-through V-model, but can be thought of as concurrent execution of processes within the V-model framework. One way to think of this is to agree that the processes that define the V-model are only required to complete in the order they appear rather than to proceed sequentially.

Systems Engineering and Lean

Lean, as I interpret it here, is the removal of low value or unneeded activities as well as the delay of significant end-user decisions until the latest possible moment. We have talked about rethinking some activities to make them more useful, and certainly most processes have some fat in them somewhere. However, delaying decision making in systems engineering is not easy. There is a drive to complete specifications, finalize allocations, and set architectural structures as early as possible.

Figure 3: *Systems Engineering as C2ISR With Spiral OODA Loop*


Observe new/updated objectives, constraints and alternatives.
• Usage monitoring.
• Competition, technology, marketplace intelligence, surveillance, and reconnaisaince.
Orient with respect to stakeholders' priorities, feasibility, and risks.
• Risk/Opportunity analysis.
• Business case/mission analysis.
• Prototypes, models, simulations.
Operate as current system.
Accept new system.
Act on plans and specifications.
• Keep development stabilized.
• Change impact analysis, preparation for next cycle (mini OODA loop).
Decide on next-cycle capabilities, architecture upgrades, and plans.
• Stable specifications, COTS upgrades.
• Development, integration, verification and validation, risk-management plans.
• Feasibility rationale.

This is especially critical when there are long lead manufacturing items in the mix. Remember, though, Lean does not delay all decisions, just those that can have significant impact on operational acceptance or high priority functionality and that can be feasibly delayed. Once you lose the early omnipotence syndrome, delayed decisions can retain design flexibility longer, enabling more rapid reaction to internal or external changes.

Systems Engineering and Team Ownership

This may be the most controversial agile attribute in a process-focused organization. If the systems engineering team owns its own process and can manipulate it to meet its project needs, how can the quality assurance folks ensure that the *correct* process is being followed? This is essentially a management decision to support empowered teams in more than name only. While it may impact the management control residing with some of the stakeholders, providing the systems engineering team with the authority and flexibility of owning their own process could radically improve their effectiveness.

Software Considerations for Agile Systems Engineering

In the introduction, I indicated that systems engineering could support Agile in other disciplines. Software is a prime example. The role of software is a significant systems engineering issue that requires adjustments, if not agility, from systems engineering processes. As systems become less *hardware with some software that helps*, and become more *software with some hardware to run on*, the need for software as a full participant in systems engineering becomes critical. This summer, the National Defense Industrial Association (NDIA) convened a group of industry, government, and academia participants to define the top problems in software-intensive systems (the majority of the systems currently built) [6]. One of the critical findings was that *fundamental system engineering decisions are made without full participation of software engineering.*

Software can no longer be relegated to a secondary activity. The days of software coders carrying out specific instructions from engineers are over. Software is what provides capability, enables flexibility, supports net-centric operations, allows quick response to new threats and environmental factors, and represents the majority of the value of a specific system, even though the hardware production may be the most expensive (and often most profitable) activity. Initial decisions must consider software architecture or they can impact the feasibility of software solutions and result in disjointed, untestable, and unmaintainable software components. The previously referenced NDIA report states the following:

> Complex, distributed, interoperating systems and evolving software capabilities have permanently altered the system level trade space. Key architectural decisions early in the system life cycle have great impact on software capabilities, attributes, and architectural/design approaches, yet the software engineering discipline is not consistently involved in these decisions.

"*While it may impact the management control residing with some of the stakeholders, providing the systems engineering team with the authority and flexibility of owning their own process could radically improve their effectiveness.*"

I like to think of this as software-first engineering. By considering software first, the SEs can take primary advantage of the flexibility and adaptability of software, define the system and its components in such a way that software development is less complex, and the system architecture and design support the effectiveness of software assurance, safety, and security. These are attributes that simply cannot be added on later, particularly in systems of systems or net-centric systems.

Final Thoughts

I have postulated that traditional systems engineering may not fit today's and tomorrow's systems because of its inherent rigidity and its often interpreted waterfall orientation. On the other hand, agility is much more a state of mind or philosophical approach than a set of rules that have to be followed regardless of appropriateness.

Despite the disagreement *from some agile proponents*, process is not the enemy – bad process is. To encourage agility, processes should not be dictated by the *process police*, but be under the control of the actors. Process experts can provide constructive support and guidance when needed, and process asset libraries should include agile or agile-friendly processes that can be used where the development environment or risk profile indicates a need for agility.

The fundamental goals of systems engineering have not changed. However, as systems grow larger and more complex, new ways of dealing with abstraction, concurrency, and uncertainty need to be developed. Agile approaches do offer reasonable and elegant ways of evolving systems and software engineering toward handling these issues.

There are still no silver bullets [7], but we can accept that there are new kinds of regular bullets available, new tactics by which they can be used, and that integrating them into our current operations can significantly improve the capability of our existing systems engineering arsenals.

As I said in the introduction, my intent with this article is to extend the dialogue about innovative ways to consider and apply systems engineering. I have not included examples, but I believe there are many systems and software engineers that have applied some of these approaches to systems engineering. I would be grateful if they joined the conversation by providing their experiences, successful or not, so that we can create better ways to balance the discipline of systems engineering with the agility required to develop today's complex defense systems.◆

References

1. Boehm, Barry, and R. Turner. Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley: Boston, 2004.
2. McCabe, R., et al. "Disciplined Agility Guidebook." Proprietary Internal Report. Systems and Software Consortium, 2006.
3. Armour, Phillip. The Laws of Software Process. Auerbach: Boca Raton, 2004.
4. Boehm, Barry, and Jesal Bhuta. USC CSSE Top 10 Risk Items: People's

Choice Awards. 2006 <http://csse.usc.edu/BoehmsTop10/>.
5. Boehm, Barry, and Jo Ann Lane. "21st Century Processes for Acquiring 21st Century Software-intensive Systems of Systems." CROSSTALK, May 2006.
6. NDIA Systems Engineering Division. "Top Software Engineering Issues within Department of Defense and Defense Industry." Aug. 2006.
7. Brooks, Frederick P. "No Silver Bullet: Essence and Accidents of Software Engineering." Computer 20.4 (Apr. 1987): 10-19.

Note

1. Mr. Krieg, Under Secretary of Defense (Acquisition, Technology, Logistics), used agile, agility, flexibility or related words nearly once a minute in a recent presentation to business executives. Mark Schaeffer, Director for Systems and Software Engineering in the Office of the Secretary of Defense, encouraged the process improvement world to become more agile in remarks at the 2006 NDIA CMMI Technology Conference.

About the Author

Richard Turner, D.Sc., a Fellow at the Systems and Software Consortium, is a researcher and consultant with 30 years of international experience in systems, software, and acquisition engineering. He is a frequent collaborator with a wide range of research organizations and system developers to transition new software-related technology to defense acquisition programs. Turner is co-author of *Balancing Agility and Discipline: A Guide for the Perplexed*, *CMMI Distilled*, and *CMMI Survival Guide: Just Enough Process Improvement.*

Systems and Software Consortium
2214 Rock Hill RD
Herndon, VA 22017
Phone: (703) 742-7116
Fax: (202) 390-3772
E-mail: turner@systemsandsoftware.org

WEB SITES

Agile Manifesto
www.agilemanifesto.com
On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, 17 people met to talk, ski, relax, and try to find common ground. What emerged was the Agile Software Development Manifesto. Representatives from eXtreme Programming, SCRUM, Dynamic Systems Development Method, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes convened. Currently, a larger gathering of organizational anarchists would be hard to assemble. The emergence of the Manifesto for Agile Software Development symbolizes the participants' intents.

Agile Advice
www.agileadvice.com
Agile Advice is a blog about agile methods such as SCRUM, Lean, and eXtreme Programming. However, it does not focus on agile software development. Rather, the focus of Agile Advice is on agile methods applied to other types of work such as managing, video-making, teamwork in general, creative working, training, writing, etc. Much of the material here is based on Mishkin Berteig's experiences as an agile coach, consultant or trainer to teams and management in organizations across North America. From time to time, other people contribute articles to Agile Advice. You are welcome to contribute as well, particularly if you have a story about agile methods, agile principles, or agile practices applied outside of software development.

The Agile Journal
www.agilejournal.com
The Agile Journal is an online magazine and monthly e-newsletter focused on providing readers with the need-to-know information and resources they need to develop software for an agile business. Among the topics covered: Open source solutions, service-oriented architecture, globally distributed development environments, Agile and iterative processes, integrated tools, and reuse and collaboration.

COMING EVENTS

May 7-11
DMSC 2007
Defense Modeling and Simulation Conference
Hampton, VA
www.ndia.org

May 7-11
PSQT 2007 West
Practical Software Quality and Testing
Las Vegas, NV
www.psqtconference.com/2007west

May 14-16
SATURN 2007
3rd SEI Software Architecture Technology User Network Workshop
Pittsburgh, PA
www.sei.cmu.edu/architecture/saturn/index.html

May 14-18
STAREAST 2007
Software Testing Analysis and Review
Orlando, FL
www.sqe.com/stareast

May 19-21
ICSP 2007
International Conference on Software Processes
Minneapolis, MN
www.icsp-conferences.org/icsp2007

May 20-26
ICSE 2007
29th International Conference on Software Engineering
Minneapolis, MN
www.icse-conferences.org/2007

June 18-21
2007 Systems and Software Technology Conference
STC Systems & Software Technology Conference
Tampa Bay, FL
www.sstc-online.org

COMING EVENTS: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

CMMI Level 5 and the Team Software Process

David R. Webb
309th Software Maintenance Group

Dr. Gene Miluk
Software Engineering Institute

Jim Van Buren
The Charles Stark Draper Laboratory

In July 2006, the 309th Software Maintenance Group (309th SMXG) at Hill Air Force Base, Utah was appraised at a Capability Maturity Model Integration (CMMI[SM]) Level 5. One focus project had been using the Team Software Process[SM] (TSP)[SM] since 2001. TSP is generally considered a Level 5 process; however, during the preparation for the assessment, it became obvious to the team that even the stringent process and data analysis requirements of the TSP did not completely address CMMI requirements for several process areas (PAs). The TSP team successfully addressed these issues by adapting their process scripts, measures, and forms in ways that may be applicable to other TSP teams.

In July 2006, the 309th SMXG was appraised at a CMMI Level 5. One of the 309th's focus projects, the Ground Theater Air Control System (GTACS) project, had been using the TSP since 2001. The team had achieved a four-fold increase in productivity during that time, had released zero defects since the TSP was adopted, and had been internally assessed at a high maturity by the group's quality assurance team. GTACS team members felt confident they could meet the rigors of a CMMI assessment and achieve their group's goal of Level 5.

Watts Humphrey, who is widely acknowledged as the founder of the Capability Maturity Model® (CMM®) approach to improvement and who later created the Personal Software Process (PSP)[SM] and TSP, has noted that one of the intents of PSP and TSP is to be an operational process enactment of CMM Level 5 processes at the personal and project levels respectively [1]. CMM and later the CMMI were always meant to provide a description of the contents of a mature process, leaving the implementer with the task of definition and enactment of these mature processes. Thus, CMM and CMMI are descriptive not prescriptive models. The TSP goal of being an operational Level 5 process implies that a team practicing TSP *out-of-the-box* should be very close to being Level 5.

The 309th is a large organization of nearly 800 employees, both civil service and contactors. The group level is comprised of five squadrons, each with a different focus or product line. 309th management and Software Engineering Process Group (SEPG) sets group policy and defines a group level process and metrics framework. Each squadron applies the group level process to its technical domain. So projects, like GTACS, must ensure their detailed project processes are consistent with their squadron's process and with group-level guidance. The GTACS project is also divided into several sub-teams, all managed as one project. The GTACS software team, which performs most of the GTACS assigned technical efforts, uses TSP to support its work. A separate Configuration Management (CM) team provides CM services. The project's customer, the GTACS Program Office, retains systems engineering responsibility and authority. This diverse organizational structure is important because several of the CMMI issues that need to be addressed are clearly the responsibility of these other entities and were not GTACS TSP team issues other than alignment and coordination.

Assessment Timeline

In order to prepare for the assessment, 309 SMXG conducted a series of Standard CMMI Appraisal Method for Process Improvement (SCAMPI[SM]) assessments which included the GTACS team. There are three kinds of SCAMPI assessments: A, B, and C. The SCAMPI A assessment is the final review during which a CMMI level can be determined. SCAMPI Bs and Cs are less rigorous and are intended to prepare the team for the full SCAMPI A. The 309th SMXG used SCAMPI Bs to ensure compliance to the model and value added to the enterprise. In general the SCAMPI B teams were told to aggressively identify risks to a successful SCAMPI A appraisal. When the SCAMPI B teams identified a process weakness, they assigned a high, medium, or low risk rating based on the seriousness of the noted weakness.

From the perspective of the TSP team there were four types of weaknesses: *non-team*, *process*, *artifact*, and *document*. The *non-team* weaknesses were those that were the responsibility of a team other than the TSP team, such as the group's SEPG or the GTACS CM team. Examples include policy changes or changes to the CM process. *Process* weaknesses indicate that the team had no process in place. An *artifact* weakness meant the assessment team found insufficient artifacts to pass the assessment. A *document* weakness meant the team's process documentation needed to be updated.

The initial SCAMPI B for the GTACS focus project was held about one year before the SCAMPI A final assessment and identified 86 weaknesses. A summary of the counts and types of these weaknesses is found in Table 1. Not all weaknesses were project focused. Some were organizational and some were squadron focused. Of the project-focused risks, many were the responsibility of one of the following: overarching project management (e.g., data manage-

Table 1: *SCAMPI B1 Noted Weaknesses*

Risk Level	Total Risks	Process Risks	Artifact Risks	Document Risks	Non-Team Risks
High	19	1	17	0	1
Medium	67	15	18	6	28
Low*	0	0	0	0	0
Total	86	16	35	6	29

* Low risks were not categorized in the first SCAMPI B

[SM] Team Software Process, Personal Software Process, PSP, TSP, and SCAMPI are service marks of Carnegie Mellon University.

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ment and stakeholder involvement plans) or the CM group. The remaining issues were the responsibility of the TSP team. Most issues were focused within the Decision Analysis and Resolution (DAR) and Causal Analysis and Resolution (CAR) PAs. The specifics of each of these are discussed in the PA section below.

Based on the results of this initial SCAMPI B, the team continued its project work. The major focus was on executing the team's CAR process and addressing the documentation and process framework issues. Significantly, the team did not devote any special resources to the CMMI preparatory effort. After this finding, preparatory work was done by the team and led by the team's process manager (a standard TSP role) as part of normal work duties. About four months into this effort the 309th realized that DAR could not be solely addressed at the organizational level and a new process requirement for DAR implementation was pushed down to the project level. The team's TSP coach developed a draft process script and team training was conducted. No opportunity to execute the DAR process occurred before the second SCAMPI B.

The weaknesses and risks identified by the second SCAMPI B are identified in Table 2. It is important to note that the assessment team for the second SCAMPI B was different than the first and that this team chose to identify areas for improvement in the low-risk areas, whereas the first team did not. These new results gave the GTACS team a different and more thorough understanding of the remaining weaknesses.

Of the weaknesses noted there were three groupings: DAR (seven High Artifact, three Medium Artifact, and one Low Document); Organizational Process Performance (OPP) (13 High Non-Team, one Medium Non-Team, and one Low Non-Team); and Training (one High Artifact, two High Non-Team, 12 Medium Document, one Low Artifact, and one Low Non-Team). The other weaknesses noted were scattered throughout the model. Of these, the most significant for the purposes of this article were the seven Medium Process weaknesses. These reflected the fact that the team had a process gap. In these seven weaknesses there were three process gaps: 1) a lack of traceability matrices in the team's engineering work packages, 2) a missing checklist item in the team's high-level design inspection checklist, and 3) the team's implementation of statistical process control (SPC) to monitor *selected subprocesses*. Of these, only the SPC issue required a major change in the team's practices. It is discussed in detail below. The team's approach to requirements traceability had previously been to include traceability information in the textual requirements, design, and test descriptions and to validate traceability via an inspection checklist item. It was straightforward to modify the engineering work package template to include the traceability tables. The missing item in the team's high-level design inspection checklist was added, although it had not caused the team issues in the past.

Risk Level	Total Risks	Process Risks	Artifact Risks	Document Risks	Non-Team Risks
High	23	0	8	0	15
Medium	38	7	6	17	8
Low	22	0	1	11	10
Total	83	7	15	28	33

Table 2: *SCAMPI B2 Noted Weaknesses*

The Software Engineering Institute (SEI) has already performed a theoretical mapping of TSP to CMMI and determined that DAR is *partially addressed* by the TSP, OPP is *supported*, Quantitative Process Management (QPM) is 90 percent *directly addressed*, and CAR is 60 percent *directly addressed* [2]. As the GTACS team set about to shore up these weaknesses, they determined that these assessments were generally accurate; they also came up with creative ways to update the TSP to completely address all of these PAs.

The PAs

In addition to the weaknesses previously described, there were also minor weaknesses in requirements management, risk management, and two QPM issues. Since the initial preparation for DAR had been only at the group level, there was no DAR process or practice in place for the project. The team's previous process improvement discussions, during their TSP post-mortems, had not produced the artifacts necessary to meet CAR requirements. The TSP post-mortem process and PSP Process Improvement Proposal (PIP) process do not require the quantitative analysis that CAR and its link to QPM does. The team had not formalized its requirements management process and its documented risks management process was not consistent with the TSP risk management process. The QPM risks were labeled as medium risks and related to a lack of thresholds and control limits.

DAR

One of the innovations the team came up with was in their approach to the Level 3 requirement for decision analysis and resolution. Initially, GTACS addressed its DAR requirements by adopting the organization's DAR processes and forms. Organizational DAR training was held for the team. GTACS created a draft operational process in the form of a TSP script. The DAR script was then used by the team to analyze three different types of issues: product design, tool selection, and process. The final DAR process was then updated and included in the team's standard process (see Figure 1, next page).

The SEI's report on TSP and CMMI identified all six DAR-specific practices as partially implemented and identified various launch meetings as points where DAR activities are implemented. We believe this *partially implemented* term underestimates the risk and resulting effort that TSP teams will face to meet DAR CMMI requirements. A better characterization of TSP's implementation of DAR is that TSP is *consistent* with DAR philosophy but is nowhere near sufficient. DAR is, at its heart, a systems engineering sub-process for making and documenting formal decisions. In some ways it is as critical to the systems engineering culture as inspections are to software engineering or personal reviews are to the PSP/TSP approach. CMMI has elevated DAR from a practice to a full-fledged PA and although TSP is consistent with DAR, TSP is insufficient to pass a CMMI assessment. A procedure like that in Figure 1 is required to produce proper and meaningful DAR artifacts.

A TSP team must also be trained in the application of DAR. Based on the background of the team members, this training may involve getting software engineers to think like systems engineers. For the GTACS team, this was surprisingly difficult. While a DAR process, like that detailed in Figure 1, may appear straightforward and obvious, software engineers may question its applicability. For years we have observed good systems engineers following processes like this to make and document their systems designs and design tradeoffs. On the contrary, it has been significantly more difficult to get

DAR Process Script

Purpose	• To guide the team in making formal decisions.
Entry Criteria	Either • A critical measurement exceeds the thresholds defined in the GTACS DAR threshold matrix. • A critical decision needing a formal analysis is identified.
General	• Critical decisions are ones that have potential impact on the project or project team. Issues with multiple alternative approaches and multiple evaluation criteria are particularly well suited for formal analysis.
Tailoring	• This procedure may be used to make and document other decisions.

Step	Activities	Description
1	Planning	- A Point of Contact (POC) is assigned. • The POC may be self-assigned if the POC is responsible for the critical decision. • The team lead assigns the POC otherwise. - The team that will perform the DAR analysis and selection activities (the DAR team) is assigned. - The POC completes the Entry section of the MXDE Decision Analysis and Resolution Coversheet (section I). - A working directory is created to hold the DAR artifacts. - An action item is created in the Project Notebook to track the status of the DAR. - The approval signatures required for this DAR are determined. • For DARs initiated because a critical measurement exceeds the thresholds defined in the GTACS DAR threshold matrix the approval signatures are documented in the Stakeholder Involvement Plan (SIP). • For other DARs the GTACS Technical Program Manager is the approval authority.
2	Identify Stakeholders	- The POC identifies stakeholders for this DAR activity. These include the following: • Those who provide the alternatives, risks, and historical data. • The DAR team. • Those who will implement the decision the DAR results in.
3	Stakeholder Input	- The DAR team obtains input from the stakeholders. • Alternative approaches. There is no limit to the number of alternative approaches identified. • Evaluation Criteria and relative weighting. • Key risks.
4	Evaluation Criteria	- The DAR team determines the evaluation criteria and relative weighting after considering the input from all stakeholders. - The DAR team reviews the evaluation criteria with the stakeholders before finalizing the criteria.
5	Selection Method	- The DAR team determines the ranking and scoring method. • Suggested ranking and scoring methods are found in the DAR Tools document. • The DAR team must agree on a scoring method, the scoring range, and have a common understanding of what the scores represent. - The selected approach is documented on the MXDE Decision Analysis and Resolution Coversheet (section II).
6	Rank Each Approach	- For each alternative, the DAR team must assign a score to each decision criteria, employing the ranking and scoring method previously selected. - The total weighted score for each alternative is determined.
7	Make a Decision	- The DAR team makes a decision and reviews it with the stakeholders making changes if necessary. - The stakeholders review is captured on the MXDE Decision Analysis and Resolution Coversheet (section III). - The final decision is captured on the MXDE Decision Analysis and Resolution Coversheet (section IV).
8	Post-Mortem	- The effort expended on this DAR is captured on the MXDE Decision Analysis and Resolution Coversheet (section IV). - Approval signatures are obtained and recorded on the MXDE Decision Analysis and Resolution Coversheet (section IV). - DAR lessons learned are captured in the DAR notes. - All DAR documents are captured and archived per the GTACS Data Management Plan (DMP). • The completed MXDE Decision Analysis and Resolution Coversheet. • Scoring and analysis worksheets. • CM is notified that the DAR is complete and that the DAR artifacts can be archived to the GTACS data management repository.

Exit Criteria	- The MXDE Decision Analysis and Resolution cover sheet is completely filled out. - The artifacts produced during the DAR activities have been archived in accordance with the GTACS DMP.

Figure 1: *The GTACS Team's DAR Process Script*

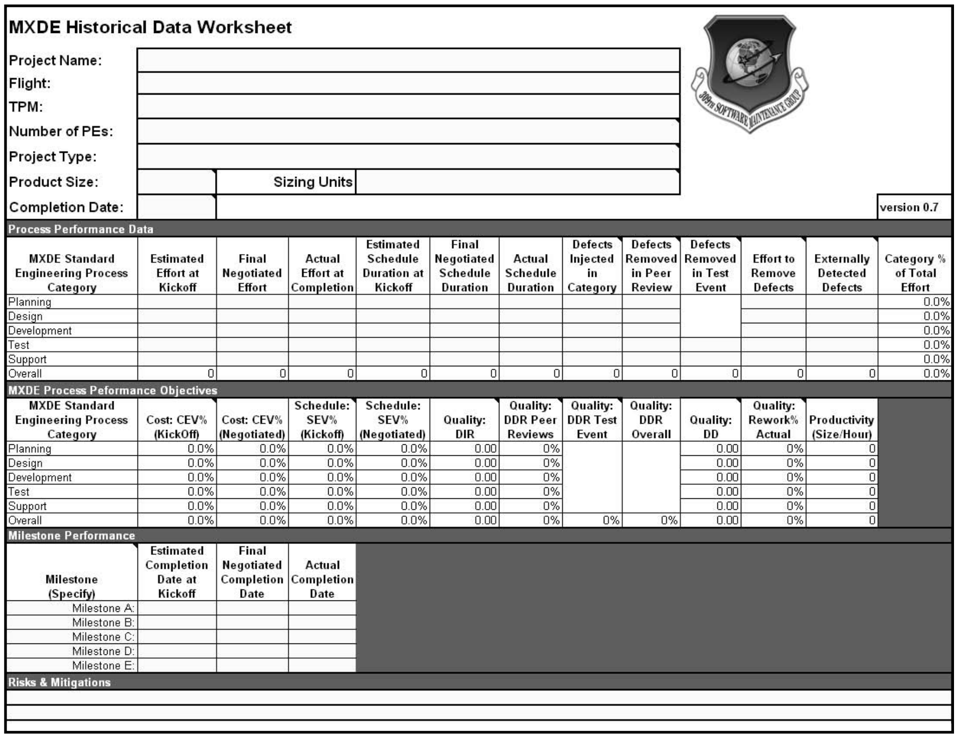
purely software engineers to document their design reasoning with the same rigor. It is, however, a basic engineering practice that can be easily learned. Our experience with the GTACS team confirmed this observation that software engineers are unfamiliar with systems engineering techniques for formal decision making and documentation but can be easily trained to use these techniques.

QPM and OPP

One contentious area surrounding CMMI High Maturity appraisals and organizations is the definition and operationalization of Maturity Level 4: Quantitatively Managed. The formative book on CMMI: *Guidelines for Process Integration and Product Improvement* describes Maturity Level 4 as the following [3]:

> **Maturity Level 4: Quantitatively Managed.** At maturity level 4, the organization and projects establish quantitative objectives for quality and process performance and use them as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance is understood in statistical terms and is managed throughout the life of the processes.
>
> For selected subprocesses, detailed measures of process performance are collected and statistically analyzed. Quality and process performance measures are incorporated into the organization's measurement repository to support fact-based decision making. Special causes of process variation are identified and, where appropriate, the sources of special causes are corrected to prevent future occurrences.
>
> A critical distinction between maturity levels 3 and 4 is the predictability of process performance.

At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are typically only qualitatively predictable.



MXDE Historical Data Worksheet

Project Name:			
Flight:			
TPM:			
Number of PEs:			
Project Type:			
Product Size:		Sizing Units	
Completion Date:			version 0.7

Process Performance Data

MXDE Standard Engineering Process Category	Estimated Effort at Kickoff	Final Negotiated Effort	Actual Effort at Completion	Estimated Schedule Duration at Kickoff	Final Negotiated Schedule Duration	Actual Schedule Duration	Defects Injected in Category	Defects Removed in Peer Review	Defects Removed in Test Event	Effort to Remove Defects	Externally Detected Defects	Category % of Total Effort
Planning												0.0%
Design												0.0%
Development												0.0%
Test												0.0%
Support												0.0%
Overall	0	0	0	0	0	0	0	0	0	0	0	0.0%

MXDE Process Peformance Objectives

MXDE Standard Engineering Process Category	Cost: CEV% (KickOff)	Cost: CEV% (Negotiated)	Schedule: SEV% (Kickoff)	Schedule: SEV% (Negotiated)	Quality: DIR	Quality: DDR Peer Reviews	Quality: DDR Test Event	Quality: DDR Overall	Quality: DD	Quality: Rework% Actual	Productivity (Size/Hour)
Planning	0.0%	0.0%	0.0%	0.0%	0.00	0%			0.00	0%	0
Design	0.0%	0.0%	0.0%	0.0%	0.00	0%			0.00	0%	0
Development	0.0%	0.0%	0.0%	0.0%	0.00	0%			0.00	0%	0
Test	0.0%	0.0%	0.0%	0.0%	0.00	0%			0.00	0%	0
Support	0.0%	0.0%	0.0%	0.0%	0.00	0%			0.00	0%	0
Overall	0.0%	0.0%	0.0%	0.0%	0.00	0%	0%	0%	0.00	0%	0

Milestone Performance

Milestone (Specify)	Estimated Completion Date at Kickoff	Final Negotiated Completion Date	Actual Completion Date
Milestone A:			
Milestone B:			
Milestone C:			
Milestone D:			
Milestone E:			

Risks & Mitigations

Figure 2: *Portion of a Standard Process Data Worksheet for the GTACS Squadron*

Assuming an organization has achieved Maturity Level 3, the concepts for Level 4 are achieved by implementing the practices and satisfying the goals for OPP and QPM. The team weaknesses identified at Level 4 in QPM and OPP were due to the facts that GTACS data was not analyzed at the sub-process level and the data analyses did not address an understanding of process variability. To understand the root cause of these issues, one must understand how standard TSP projects use data to quantitatively manage themselves.

TSP uses data for three purposes: project planning, project monitoring and oversight, and process improvement. For project monitoring, TSP fundamentally considers the software development process as a single entity whose purpose is to help guide the production of products. Earned Value (EV), TSP's primary tool for analyzing schedule and cost, measures the whole process and not subprocesses. TSP's two primary tools for monitoring quality, Percent Defect Free (PDF) and Process Quality Index (PQI) also do not focus at the sub-process level. PDF considers the whole product and the whole process. PQI focuses on the evolving quality of product parts by analyzing the whole process used to produce them. Its usual use is to identify potentially troublesome parts for additional quality analysis. In addition, none of these measures consider variability from the statistical process control perspective. EV considers only how actual cost and schedule performance is varying from the planned performance. Both PDF and PQI consider how quality performance varies from TSP supplied benchmarks.

OPP looks at quantitative management from a top-down perspective. After the organization determines the critical processes (or subprocesses) and associated measures, analysis procedures, and performance models, a project can then use the practices of QPM to fulfill project OPP requirements. The organization's OPP requirements define the key organizational metrics as cost performance index, schedule performance index, yield, and rework. The team's base TSP practices are collecting all the measures needed to meet these requirements. Figure 2 is a portion of the squadron's historical data worksheet showing the key measures the project must collect and submit and the key metrics derived from those measures.

As noted earlier, the SCAMPI B assessment team had identified the team's use of EV and PQI (the team was not using the TSP PDF metric because it did not add value for its work) as possibly not fulfilling the intent of the variability of subprocesses clauses of QPM. After discussion, the team decided to track rework and the forecast completion date of its various work products. These also supported the team's two highest priority project goals: finishing its work on time and having low rework. The key selection criteria for these two metrics were that they could be tracked during the project, that corrective action could be taken if they were trending beyond limits or goals, and that they were of relatively low cost to implement.

The team's EV tool computed the forecast completion date of the project and because of the way the project plan was set up, it could also compute the forecast completion date of each of the project subparts. The team reviewed these forecasts at the subpart level every week. Only once, when a team member had a medical condition that required unplanned long-term leave did a forecast fall past the project end date, causing the team to replan its approach for this particular subpart. This matches our prior TSP experience where the TSP EV project tracking process leads the team to meet its schedule commitments [4].

The team was easily able to use rework in a way that satisfied the CMMI assessor's need to see the team reviewing process variability. Rework time for this TSP team was defined as time recorded in the defect logs. Percentage rework was rework time divided by total task time. Good historical data existed from the team's prior projects. Rework percentage was computed weekly and reviewed during the team's weekly meeting for both the project's subparts and the project as a whole. Rework remained within control limits throughout the entire project for all project parts. Figure 3 (see page 20) is the project-level rework plot that was reviewed by the team during its weekly meeting. The rework percentage for each of the team's subparts and the project as a whole were each plotted. The plots each included the subpart or project under review, the organizational goal (10 percent), the Upper Control Limit (10.46 percent), and the normalized (to the project schedule) plots for previous projects.

The good news is that the data collection required by the TSP provides all and more of the data needed to perform such analyses. Using these data, the GTACS project was able to come up with QPM analyses that focused on variability for effort, schedule, and quality performance (such as rework) within predicted parameters. The team updated their weekly meeting process to address each of these measures, to see if they were in control, and to bring items that had gone astray back under control. GTACS also added items to the TSP post-mortem process to collect project closeout data that could be used to determine process performance and variability overall and at the sub-process levels. These data were then standardized for sharing across the organization, supporting the requirements of OPP (Figure 3).

CAR

The TSP process as it currently stands calls for a detailed post-mortem analysis of project and process data, including identification of improvements. This provides a great deal of support for the Level 5 CAR requirement; however, the TSP does lack CAR formality and feedback to determine if implemented process improvements really worked. In order to shore up these issues, the GTACS team updated the post-mortem script to directly address CAR. They created a requirement for a *CAR report*, which formally douments the TSP post-mortem by capturing the data analyses performed, weaknesses identified, and the suggested process changes to address these weaknesses. The report also adds to the TSP post-mortem an analysis of the impact of previous process improvements.

Training

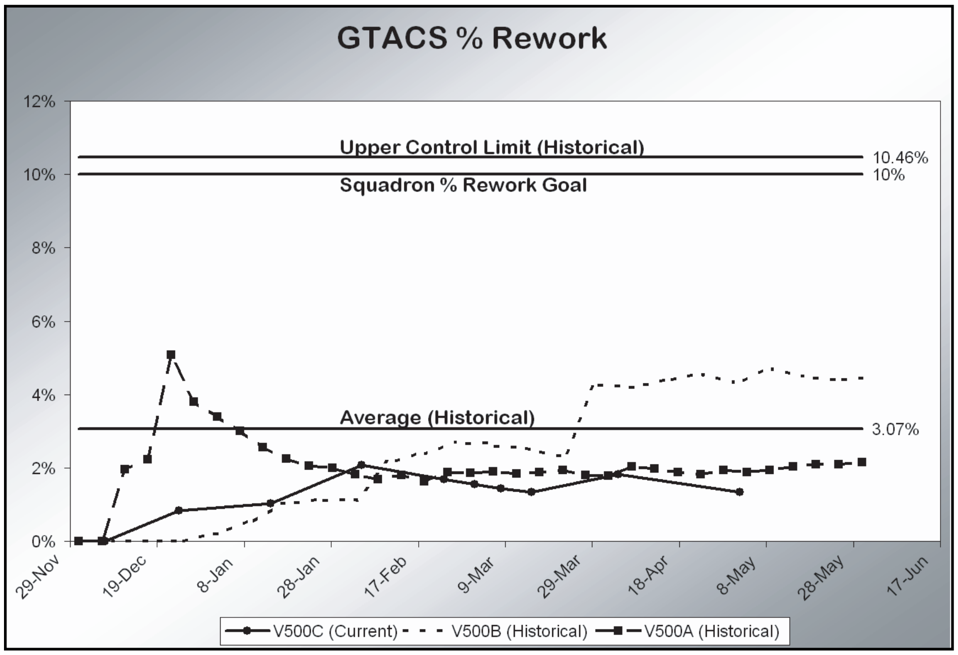
The TSP rollout strategy that the GTACS team used included PSP training for all engineers and managing TSP teams training for the team lead and the GTACS TPM. This approach provided the primary training for eight of the 21 PAs. Additional organizational specific training on policy was still required. The PAs addressed by PSP/TSP were project planning, project monitoring and control, integrated project management, integrated teaming, process and product quality assurance, measurement and analysis, and CAR. Verification was partially addressed. Training was required for the management PAs of risk management and quantitative project management, all the engineering PAs (Requirements Development, Requirements Management, Technical Solution, Product Integration, Validation, and Verification), the support PAs, configuration management, and DAR, and all the process management PAs (Organizational Process Focus, Organizational Process Definition, Organizational Process Performance, and Organizational Innovation and Deployment).

The team addressed the training issue by creating a team training plan that discussed how new team members acquired the skills needed to become full team members. This included an approach to obtaining GTACS domain knowledge, the tools and technologies used by the team, the processes used by the team, and the key organizational training needed to support the team. Most of the details of these training packages had been in existence for several years but were not structured and organized. In fact, the team had a longstanding improvement proposal to organize its training approach.

Summary

The GTACS team in 309th SMXG at Hill Air Force Base, Utah, successfully used the TSP in reaching their goal of CMMI Level 5. In order to do so, they adapted from and added to the TSP scripts, measures, and forms in ways that they believe can help other TSP teams also achieve this feat, as far as can be done by a single focus project.◆

Figure 3: *Variability in Rework as Tracked by the GTACS Team*



Related Literature

The topic of relating TSP practice to CMM-based assessments has been addressed in two thought papers and at least two case studies. The thought papers studied the problem in the abstract by comparing a theoretical TSP project against a model. Davis and McHale [5] first compared TSP against the CMM and concluded that *TSP implements a majority of the key practices of the SW-CMM*. McHale and Wall [2] later extended this study to the CMMI. They concluded, *that TSP can instantiate a majority of the project-oriented specific practices of CMMI*.

Naval Air Systems Command used TSP to advance their CMM efforts. Their experience report compared their approach of using TSP to implement CMM improvement versus non TSP based CMM improvement approaches. They reported that they halved the time needed to move from CMM level 1 to CMM level 4 by basing their process on TSP [6, 7]. Cedillo reported that *TSP actually accelerates CMM/CMMI implementation in a small setting* where the process improvement approach of a small startup company was based on TSP [8].

References

1. Carnegie Mellon University (CMU). TSP and CMMI: A Brief History. <www.sei.cmu.edu/tsp/history.html>.
2. McHale, James, and Daniel S. Wall. "Mapping TSP to CMMI." CMU/SEI-2004-TR-014. CMU, 2004.
3. Chrissis, Mary Beth, Mike Konrad, and Sandy Shrum. "CMMI®: Guidelines for Process Integration and Product Improvement." CMU, 2003.
4. Webb, David. "All the Right Behavior." CrossTalk Sept. 2002 <www.stsc.hill.af.mil/crosstalk/2002/09>.
5. Davis, Noopur, and Jim McHale. "Relating the Team Software Process℠ to the Capability Maturity Model® for Software." CMU/SEI-20020TR-008. CMU, 2002.
6. Pracchia, Lisa. "The AV-8B Team Learns Synergy of EVM and TSP Accelerates Software Process Improvement." CrossTalk Jan. 2004 <www.stsc.hill.af.mil/crosstalk/2004/01>.
7. Wall, Daniel S., James McHale, and Marsha Pomeroy-Huff. "Case Study: Accelerating Process Improvement by Integrating the TSP and CMMI." CMU/SEI-2005-SR-012. CMU, 2005.
8. Cedillo, Karina. "Accelerating CMMI Implementation With PSP and TSP in a Small Organization." SEPG, 2005.

About the Authors

David R. Webb is a senior technical program manager for the 309th Software Maintenance Group at Hill Air Force Base in Utah, a CMMI Level 5 software organization. He is a project management and process improvement specialist with twenty years of technical, program management, and process improvement experience on Air Force software. Webb is an SEI-authorized instructor of the PSP, a TSP launch coach, and has worked as an Air Force manager, SEPG member, systems software engineer, and test engineer. He is a frequent contributor to CROSSTALK, and holds a bachelor's degree in electrical and computer engineering from Brigham Young University.

309 SMXG/520 SMXS
7278 Fourth ST
Hill AFB, UT 84056
Phone: (801) 940-7005
DSN: 775-3023
Fax: (801) 775-3023
E-mail: david.webb@hill.af.mil

Gene Miluk, Ph.D., is currently a senior member of the technical staff at the SEI in the Acquisition Support Program. For the past 15 years, he has worked with client organizations undertaking software process improvement, software acquisition improvement and technology transition. Prior to joining the SEI, he was the founder of the Denver Metrics Group, Inc. a firm specializing in software measurement and metrics. He has been a frequent lecturer on software measurement and software process improvement as well as an instructor for The University of California at Berkeley, The University of California at Irvine, The University of Colorado at Denver and both Graduate School of Industrial Administration and The Heinz School at CMU. He holds a degree in systems engineering from the Polytechnic University of New York, a masters in information systems from the University of Colorado, and a doctorate in organizational change from Pepperdine University.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213
Phone: (412) 268-5795
E-mail: gem@sei.cmu.edu

Jim Van Buren provides support to the Air Force's Software Technology Support Center (STSC), where he brings over 25 years of software development and management expertise to STSC customers including the GTACS software team. He is an SEI-authorized PSP instructor and TSP launch coach. Van Buren is on the technical staff of the Charles Stark Draper Laboratory, serving as Draper's STSC program manager. He holds a Bachelor of Science in computer science from Cornell University.

Charles Stark Draper Laboratory
517 Software Maintenance Squadron
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056
Phone: (801) 777-7085
Fax: (801) 777-8069
E-mail: jim.vanburen@hill.af.mil

"OO-OO-OO!" The Sound of a Broken OODA Loop

Dr. David G. Ullman
Robust Decisions Inc.

The Observe, Orient, Decide, and Act Loop (OODA) was developed to describe the process needed to win at war. Recently, the OODA Loop has been applied to business and product development as a way to describe decision-making cycles. In these situations, the loop often gets stuck at the D, and the team is reduced to making a sound like OO-OO-OO. This article explores why it gets stuck and how to put the D in the loop as a basis for effective action.

Col. John Boyd, U.S. Air Force fighter pilot ace, developed the concept of the OODA Loop to describe the process needed to win at war. This model matured as he won aerial dogfights in Korea and Viet Nam and later used it to describe how to gain a competitive advantage in any situation. Recently, the OODA loop has begun to be applied to business and product development as a way to describe their decision-making cycles. In these situations, the loop often gets stuck at the D and the team is reduced to making a sound like *OO-OO-OO*[1]. The OODA loop is a succinct representation of the natural decision cycle seen in every context: war, business, product development, or life.

Boyd diagramed the OODA loop as shown in Figure 1. In words, all decisions are based on observations of the evolving situation tempered with implicit filtering based on the problem being addressed. These observations are the raw information on which the decisions and actions will be based.

The observed information needs to be processed to orient it for further making a decision. In notes from his talk *Organic Design for Command and Control*, Boyd said:

> The second O, orientation – as the repository of our genetic heritage, cultural tradition, and previous experiences – is *the most important* part of the OODA loop since it shapes the way we observe, the way we decide, the way we act. [1]

As stated by Boyd and shown in the Orient box, there is much filtering of the information through our culture, genetics, ability to analyze and synthesize, and previous experience. Since the OODA loop was designed to describe a single decision maker, the situation is usually much worse than shown as most business and technical decisions have a team of people observing and orienting, each bringing their own cultural traditions, genetics, experience, and other information. It is no wonder that we often get stuck here, and the OODA loop is reduced to the stuttering sound of OO-OO-OO.

Getting stuck means that there are no decisions and thus no actions. In reality, a decision has been made to do nothing. Time keeps moving, and resources are used. In Boyd's warfighter scenario, the enemy gets the upper hand. In business, the competition keeps progressing in its OODA loops while you keep using your resources while adding no value. In other words, getting stuck at the decision point can have severe, even grave, consequences.

The organizational response to being stuck is often more analysis, more data, more simulations, or more *decision by wringing hands*. Sometimes these efforts help, if directed at the right *sticking point*, but often these activities only postpone decisions until some external event occurs that demands a decision. This results in *decision by running out of time* or, if the action is dictated by a superior, *decision by fiat*. Neither of these have much chance of being a robust decision.

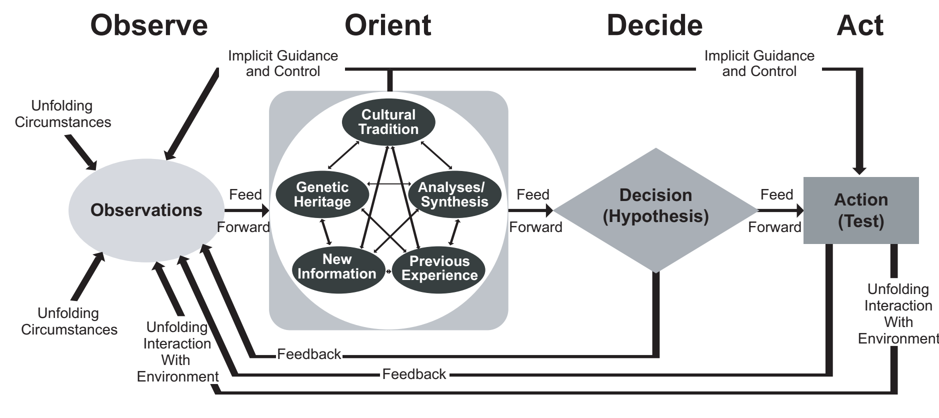
An important feature of the OODA loop is that it is not static, it is a loop. Efforts at orientation affect what is observed and how the actions are implemented. Each decision and action changes the context for the observations, and the result of the action on the environment causes a push-back that affects the information being observed. Competitive advantage comes from quickness over the entire *loop,* and, as with each iteration, the changes are smaller (as they are modifications to an understood situation) and can be more easily managed – therefore staying ahead of the competition.

To explore why we get stuck, consider the expanded OODA loop in Figure 2.

In this diagram, the OODA loop elements are detailed as activities that are keys to success. The dark box around orient and decide emphasizes where the bulk of the discussion is focused. In the following, think of each task in a project or the development of each feature in a product as an OODA loop.

Observations originate from human sources as well as from data, test results, intelligence sources, and models about a situation. In software and product development, observations include the following: formal specifications developed by the customer; competition's products; the results of data collection; and the incomplete and evolving results of other projects. Regardless of the observation source, this information is *evolving*, *inconsistent*, *uncertain*, *incomplete*, and is *dependent* on who is doing the observing (e.g. two intelligence sources may give conflicting information, or two engineers may interpret the results of a simulation differently). Further, some of the information is qualitative and some is quantitative. This informational mess is characteristic of most critical combat, technical, product development, and business situations. The goal

Figure 1: *OODA Loop*



of orient is to reduce this mess so we can decide what to do next and take action – collect more information, involve more people, or turn our attention to other OODA loops.

The goal of *orientation* is to *make sense* of the observations. This requires understanding the observations as a basis for choosing the best course of action. In many cases, formal analysis can help reduce this fog, but much of the information cannot be easily modeled. Thus, how this information is managed to match the human decision-makers' needs is crucial.

Orientation also is dependent on viewpoint. Even on the same team, how the observations are understood is dependent on who is trying to understand them. As Boyd pointed out, understanding is dependent on previous experience, cultural traditions, and genetic heritage. Beyond these measures, understanding is also dependent on one's role in the organization and team objectives. Helping a team make sense of the situation and develop a shared understanding while honoring the different viewpoints is a challenging but necessary part of getting team buy-in and making a robust decision.

Orientation should aid in the *sharing of implicit knowledge*. By this we mean that in trying to make sense of the situation and fuse the observations, some of the stakeholder's implicit knowledge must become explicit and be communicated to others.

Often the OODA loop stalls because the decision makers are not comfortable with the uncertainty. *Managing uncertainty* implies that beyond concern there is an effort to do the following: measure the uncertainty, control what you can, and minimize the effect of that which you cannot control. Uncertainty creates risk that a poor decision will be made. This is over and above traditional risk consideration – risk based on past statistics that give information on the probability of occurrence and consequence. Since decisions require a look into the evolving future, traditional probability methods (often called frequentist methods) for managing risk and uncertainty cannot be applied. Recently, Bayesian methods have been used to help manage these situations (see item in 4c, to follow).

A key part of orientation is *developing alternative courses of action*. In the words of the French philosopher Emile Chartier, "Nothing is more dangerous than an idea when it is the only one you have." In engineering design and software development, this means actively searching for multiple options to consider.

Making a *decision* is not a single action,

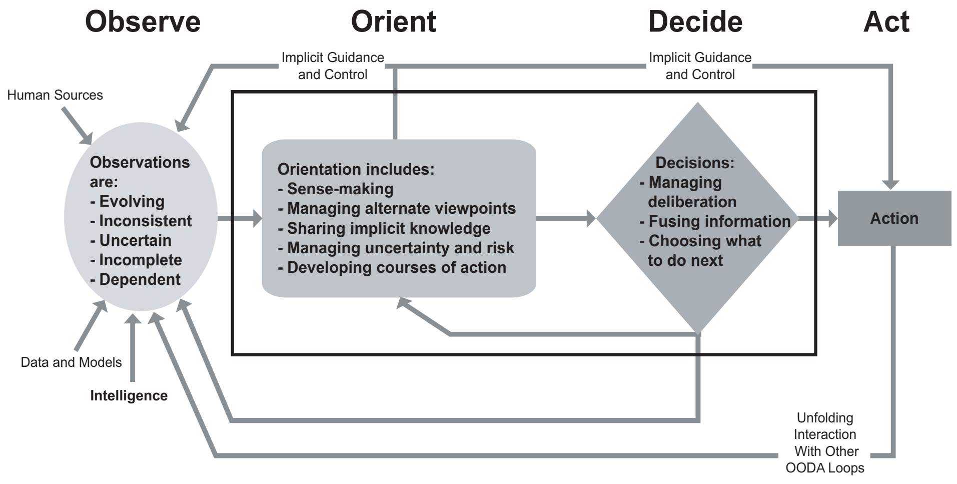

Observe
Orient
Decide
Act
Implicit Guidance and Control
Implicit Guidance and Control
Human Sources
Observations are:
- Evolving
- Inconsistent
- Uncertain
- Incomplete
- Dependent
Orientation includes:
- Sense-making
- Managing alternate viewpoints
- Sharing implicit knowledge
- Managing uncertainty and risk
- Developing courses of action
Decisions:
- Managing deliberation
- Fusing information
- Choosing what to do next
Action
Data and Models
Intelligence
Unfolding Interaction With Other OODA Loops

Figure 2: *Expanded OODA Loop*

but is a process of repeatedly *deciding what to do next – observe* more information, do further *orientation*, or take *action*. A major component of this is *managed deliberation*, which is synergistic with Orient, as it is part of sense-making and can help lead to a shared vision of observations. Managed deliberation implies the following:

- Identifying the areas on which to focus based on benefit of further effort. This is a major sticking point in the OODA loop. It is often difficult to see where more work needs to be focused. The benefit is usually hard to measure, but it should be in terms of the following: 1) anticipated change in satisfaction with a course of action, 2) anticipated change in the risk with a course of action, or 3) anticipated consensus or buy-in from management or team members.
- Identifying the cost of further effort. This also is a major sticking point in the OODA loop. The cost of doing more work is usually in terms of the time used and the expense for researching, testing, or consulting.
- Identifying areas where consensus is low and impact is high. The goal here is to separate areas that need effort (consensus is low and impact is high) from points that are not critical to a decision. Part of choosing what to do next is separating out what is easy to do from what will actually provide understanding needed to make a decision.
- Managed deliberation implies OODA loops inside OODA loops as the decision about that to work on next requires its own OODA activities.

Deciding what to do next requires *fusion* of the orientation results. As with the observations, the result of orientation is usually evolving, inconsistent, incomplete, uncertain, and dependent on who is doing the orientation. Somehow, this oriented information must be fused to develop a picture of the situation that is cognitively small enough to decide what to do next.

Fusion may be both an analytical effort and a consensus building effort. Analytical methods range from formal optimization to methods that combine the subjective opinions of team. More importantly is building collaboration to get buy-in on the chosen action. Collaboration requires that the following is present:

- Everyone can paraphrase the issue to show that it is understood.
- Everyone has a chance to contribute to the solution of the problem.
- Everyone has a chance to describe what is important.

Those who do not agree with the final decision will more likely support the team because they have been included in the decision-making process and appreciate the compromise needed to reach a decision

The proof of the success of the OODA loop is in the success of the Action taken. Here, think of actions as work activity or pieces of information that affect work activity. All action affects future observations. In *Why Decisions Fail*, [2] the author studied 400 decisions made by senior managers in medium to large organizations. He considered the decision a success if it was sustained for two years after the decision was made. In other words, the action taken had noticeable impact two years later. He found that fully half of the decisions failed to have any impact beyond the use of resources.

It is clear that many decisions in information technology OODA loops fail. According to the 2004 Chaos Report [3], 53 percent of products are delivered late or over-budget, and an additional 18 percent are cancelled. Further, projects completed by large companies have only 42 percent of the originally designed features and functions. Features and functions are often jettisoned during a project to help meet schedule and budget. This is often referred to as *descoping* a project; some organizations build descoping into their original plans. The Chaos Report numbers may be worse than stated as they are self-reported.

Guidelines for Unsticking the OODA Loop

As ubiquitous and important as the OODA loop is, most of us receive little training in how to perform the two key elements of orient and decide. Sure, we pick up some clues from our formal training, yet we are never formally trained in the OODA elements. Even in military training [4], there is little detail about how to manage them. Itemized here are a few guidelines for staying unstuck and for making robust decisions, especially in product and software development.

The Entire OODA Loop:

1a. Identify the OODA loops in your organization and their interactions. Each OODA loop provides the environment for other interacting OODA loops. Consider each task or feature development as an OODA loop and think through O-O-D-A.

1b. For each OODA loop, ensure you know who the resulting actions will affect because they, in turn, may affect your observations as your loop is refined.

Observe

2a. Make sure you know the properties of *observations*. Each piece of information comes with details about its stability, consistency, certainty (see 2b), completeness, and its dependence on the observer. Note these and formalize them, if possible.

2b. All *observations* are uncertain. Early in the design of a system, uncertainty is dominated by lack of knowledge – cognitive uncertainty. As systems mature, most uncertainty is due to natural variance in the environment and nature of materials. In software development, variation is small compared to cognitive uncertainty. Anytime anyone gives you an estimate, the results of a simulation or experiment, or an opinion, you must tag it with a level of certainty. You need to make this explicit. Engineers and financial analysts in particular are prone to giving single, deterministic values for information that is really a distribution. Push back on them to find the distribution, even if it is in terms such as *very sure*, *about*, or *sort-of*. Early in the development of a system, all estimates are uncertain and need to be managed as such (see item 3d).

Orient

3a. Since *orientation* is so important, it is amazing that more emphasis is not put on its component parts. The major function of orientation is making sense of the observations. Since all observations are understood only in relation to what the orienter knows; *sense* is different to each person presented with the observations. Thus, one sticking point is when the person responsible for the OODA loop does not have sufficient knowledge to orient and knows it. This realization may take awhile. Thus, if responsible for a decision and it is not happening, ask if it is because of insufficient knowledge to make sense of the situation. If so, find people who do have the knowledge.

"... understanding is dependent on previous experience, cultural traditions, and genetic heritage. Beyond these measures, understanding is also dependent on one's role in the organization and team objectives."

3b. If a problem is sufficiently complex that a team is involved, then each person on the team has a different context for *orienting*. Here, sense making is communal and challenging so no one person has either a complete picture or the capability of developing one. It is possible to have meetings to discuss the observations without significant sense making. The key is to set up environments that support sense making by sharing pertinent information needed for the decision. Implicit knowledge needs to be made explicit in a form that is understandable by others who have a different context for understanding the observations.

3c. In a team situation, during *orientation*, there will be multiple viewpoints about what is important. It is necessary to separate what is important from what is to be achieved. For example, the cost of an alternative may be very important to some and not as important to others. This fact needs to be separated from the estimated cost of each alternative being considered. The uncertainty in the estimate may swamp the differences in importance, but only if this separation is made explicit. To restate this, separate out what is to be achieved (i.e. goals, targets) from how important it is to achieve them. Further, disagreements about what is important can be an asset as management of them can support collaboration leading to *action* buy-in.

3d. Since observations are uncertain, *orientation* methods need to be able to manage uncertain information whether quantitative or qualitative. The risk of not making a robust decision is dependent on managing this uncertainty. One way to tackle uncertainty in software development is through simulation and testing across the range of the uncertainty. This has been formalized through the use of design of experiments (DOE) and Taguchi methods [5].

3e. During *orientation*, make sure you are considering multiple courses of action and can itemize them. Develop methods within your organization that encourage this. Find ways to help the champions of each idea compare and contrast their alternatives with others.

Decide

4a. Making a *decision* is essentially deciding what to do next. The default is to do nothing – getting stuck on OO-OO-OO. Being stuck is a clear call for any of the following:

- Build consensus with the information you have. This pushes back on orientation – managing viewpoints, sharing implicit knowledge, collaborating, and developing new courses of action. This is the first choice about what to do as it is the most cost effective.
- Perform more analysis to refine the orientation information. This is generally more expensive than working with the information you have and can lead to *paralysis by analysis* – the risk-averse activity of trying to drive out all uncertainty by undertaking increasingly higher-fidelity simulations of the situation. When the fidelity of the simulations is superior to the certainty

of the observations on which the simulations are based, time and money are being wasted.
- Return to observation and collect more information. This is almost always more expensive and time consuming than the previous two options. If the information that will reduce the risk and unstick the decision is collectable, it may be worthwhile.

4b. Work toward learning from past *decisions*. Knowing how well you are doing requires keeping track of decisions made, the actions that follow, and the success of the actions (i.e. did they stick?). This is seldom done in a fashion that makes it possible to learn from OODA loop successes and failures.

4c. Develop methods that manage the fusion of uncertain observations and orientation in support deciding what to do next. Formal tools that help you do this are just being developed. Since decisions are based on uncertain estimates of the future, Bayesian methods are ideal for supporting such activities [6]. In one such effort developed by the author, Bayesian tools are packaged in a distributed team decision-support system. In this system, there is no need to understand the Bayesian mathematics that are hidden behind an easy-to-use graphical user interface. This system attempts to estimate the risk of making a poor decision and, in many ways, supports the management of the uncertain observations and orientation.

Act

5a. A decision that has both high buy-in and accountability naturally generates actions that are aligned with the decision made. The opposite is also true. If a decision is made and it is not followed by consistent actions, then the problem may lie in earlier OODA activity (especially see items 3b, 3c, 4a, and 4c).

5b. Associate the actions taken with specific OODA loops (e.g. tasks). If you cannot identify where an action initiated then it may be an assumption that has no formal OODA activity behind it and may be spuriously driving other loops. Think of actions as any work effort or piece of information that is affecting work effort.

In summary, the OODA loop model is an easy way to think about your product development effort. It can help focus on problems that occur along the way – especially if you hear your organization stuttering OO-OO-OO. Following these guidelines can help unstick your OODA loops.◆

References

1. Boyd, J. "Organic Design for Command and Control" <www.d-n-i.net/boyd/pdf/c&c.pdf>.
2. Nutt, Paul C. Why Decisions Fail. Berrett-Koehler Publishers, 2002.
3. The Standish Group. "2004 Third Quarter Research Report: CHAOS Demographics." <www.standishgroup.com/sample_research/index.php>.
4. "Commander's Estimate of the Situation." NWC 4111e. Mar. 2002. Naval Operations Planning, Naval Warfare Publication May 2001 (formerly NWP11).
5. Phadke, Madhav. "Design of Experiments for Software Testing." iSixSigma Magazine Jan. 2003 <www.isixsigma.com/library/content/c030106a.asp>.
6. D'Ambrosio, Bruce. "Bayesian Methods for Collaborative Decision-Making." Robust Decisions <www.robustdecisions.com/bayesianmethoddecisions.pdf>.

Note

1. The sound "OO-OO-OO" was verbally described in a presentation by Harvey S. Gold, lead design for Six Sigma and Black Belt, DuPont CR&D, June 2005.

About the Author

David G. Ullman P.E., Ph.D., (Emeritus Professor of Design, Oregon State University) has researched design and decision making for more than 25 years. Recently, his research has focused on the importance of decision-making in the product realization process. He is an active software and hardware designer. His recent book, *Making Robust Decisions*, was published in December, 2006. Additionally, he is the author of *The Mechanical Design Process*, 3rd edition. He is an American Society of Mechanical Engineers Fellow and a registered Professional Engineer.

Robust Decisions, Inc.
1655 Hillcrest DR
Corvallis, OR
Phone: (541) 758-5088
(541) 754-3609
E-mail: ullman@robustdecisions.com

Using Switched Fabrics and Data Distribution Service to Develop High Performance Distributed Data-Critical Systems

Dr. Rajive Joshi
Real-Time Innovations, Inc.

High performance and predictability are prerequisites for any large-scale networked system dependent on real-time data processing and analysis. Data representing actual events or system status must be evaluated while it is still relevant to tactical conditions, making it imperative to know when specific data is available to aggregate and evaluate that data in real time. Unreliable receipt times make effective analysis difficult or impossible.

Fast and predictable performance is always an issue in the design of a large-scale networked system dependent on real-time data processing and analysis, but especially so when designing distributed systems with thousands of nodes that need to move a lot of data around quickly in a dynamically changing environment. Switched-fabric networks [1, 2] can provide fast and highly scalable hardware solutions and are now being increasingly used in such applications. What is needed beyond that is a software solution for bringing predictability, flexibility, and reliability to distributed data communications. I describe how the Data Distribution Service (DDS) [3] data-centric publish-subscribe middleware layer can realize the full potential of a hardware switched fabric network to deliver a complete solution for application developers.

Data-Critical Systems Share Characteristics

Many large-scale, data-critical applications can be characterized by three attributes: the need to gather and distribute data in real-time, the large amount of data being transferred, and the entities involved in this data exchange are varied and may even change over time. For instance, air traffic control, financial transaction processing, battlefield, naval command and control, or industrial automation systems all are examples of data-critical systems which have these three attributes.

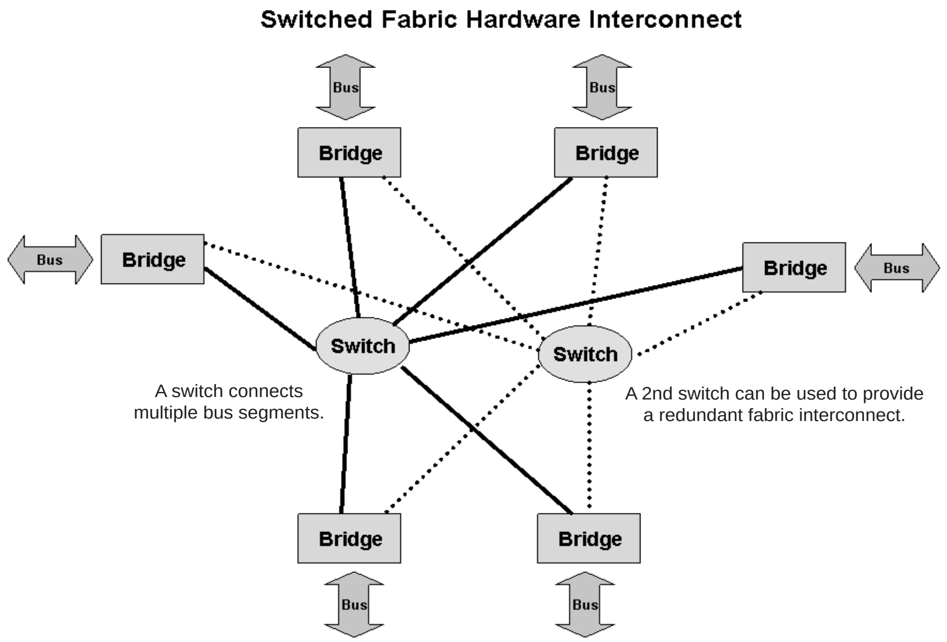
These systems are not necessarily hard real-time, but their predictability requirements are an integral part of the functions they perform. They gather data from a variety of sources, sensors for example, and they distribute the data to a variety of users like databases, display devices, or control algorithms. Furthermore, by their very nature, they are distributed.

Today's bus-based architectures, typically multi-Central Processing Units (CPU), Versa Module Europa (VME) backplane solution with hard-wired input/output (I/O) interfaces to sensors and effectors, fall short in several areas in addressing the needs of data-critical systems. For example, these hardware transport mechanisms do not scale, are difficult to make fault-tolerant, and are difficult to modify and upgrade once they have been deployed.

For these reasons, designers of complex, data-critical distributed systems are turning to switched fabrics to replace bus backplane and serial interconnect technologies. StarFabric, Peripheral Component Interconnect (PCI) Express Advanced Switching, Serial Rapid I/O and InfiniBand are some commercial products that implement different switched fabric designs [1, 2].

A switched-fabric bus is unique in that it allows all nodes on a bus to logically interconnect with all other nodes on the bus (Figure 1). Each node is physically connected to one or more switches. Switches may be connected to each other. This topology results in a redundant network or fabric, in which there may be one or more redundant physical paths between any two nodes. A node may be logically connected to any other node via the switch(es). A logical path is temporary and can be reconfigured, or switched among the available physical connections. Switched fabric networks can be used to provide fault tolerance and scalability without unpredictable degradation of performance, among other features.

Figure 1: *Switched Fabric Architecture. Multiple Switches Can Be Used to Expand the Fabric and Provide Hardware Redundancy*


Switched Fabric Hardware Interconnect
Bus
Bridge
Switch
A switch connects multiple bus segments.
A 2nd switch can be used to provide a redundant fabric interconnect.

Switched Fabrics and Data Distribution Service

A key characteristic of switched fabrics is that they allow peer-to-peer communication between nodes without having to physically connect every node to every other node. With every node physically connected to every other node, adding a new node is exponentially more and more expensive as the number of nodes increases. Because a switched fabric network employs switching to achieve logical connectivity and reconfigurability, these systems can be architected to be highly scalable.

On the software side, publish-subscribe communication systems naturally map onto switched fabrics. Publish-subscribe systems work by using endpoint nodes that communicate with each other by sending (publishing) data and receiving (subscribing) data anonymously via topics. A topic is identified by a name and a data type. A data producer declares the intent to publish data on a topic; a data consumer registers its interest in receiving data published on a topic. The middleware acts as the glue between the producers and the consumers; it delivers the data published on a topic by a producer to the consumers subscribing to that topic. There can be as many topics as needed – a producer can publish on multiple topics and a consumer can subscribe to multiple topics. The middleware layer isolates the data producers from the consumers; they have no knowledge of each other.

A publish-subscribe software architecture allows producers and consumers to be loosely coupled. As a result, it is naturally scalable and can easily adapt to the changing needs of distributed data-critical systems. The producers and consumers are peers – they directly communicate with each other, so the topology of publish-subscribe systems can be closely matched to that of switched fabric systems. Thus, a publish-subscribe middleware layer can fully exploit the potential switched fabric network hardware.

DDS standard (see The DDS Standard sidebar on page 28) specifies a data-centric publish subscribe middleware layer, developed with the needs of distributed data-critical applications in mind. A well-designed DDS middleware implementation can be good at real-time data distribution, be easily field-upgradeable, and be transport agnostic. It can be better at real-time data distribution because publish-subscribe is more efficient than the traditional request/reply based architectures in both latency and bandwidth for periodic data exchange. Further, applications can be easier to upgrade in the field because publishers and subscribers do not care who or how many their counterparts are. And finally, since the middleware is layered on top of the physical means of getting the data from one place to another, it does not need to depend on the network transport or topology used.

Figure 2 illustrates the DDS data-centric publish-subscribe architecture. A topic has a name and a data type associated with it and represents the application data model. DataReaders and DataWriters are associated with topics. A DataWriter can publish data on its associated topic; a DataReader can subscribe to data on its associated topic. DDS middleware automatically and anonymously sets up direct data flows between DataWriters and DataReaders associated with a topic, resulting in scalable and fault-tolerant data distribution.

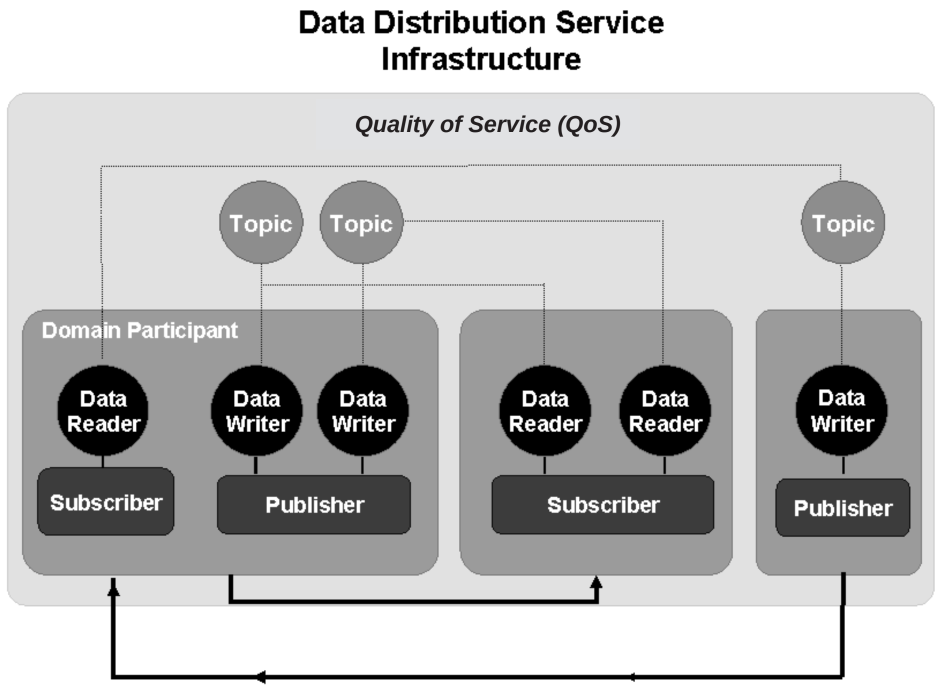


Figure 2: *DDS Data-Centric Publish-Subscribe Architecture Organizes the Data in a Distributed System Around Topics*

New Choices for System Architects

This marriage of switched fabrics and DDS real-time middleware offers architects new flexibility in adding capabilities that were once much more difficult to achieve. Many of the features offered by switched fabrics have complementary capabilities in the DDS-compliant middleware. For example, switched fabrics typically offer rich error management features such as the ability to recognize, report, and route around failed paths. With DDS-compliant software, system designers can also take advantage of DDS error reporting facilities.

A key feature of switched fabrics is support for multiple paths between nodes. This gives system architects the ability to easily implement multiple physical interconnects that can be combined with sophisticated error management. Likewise, with DDS, applications can take advantage of redundant publishers that have different strengths. When a higher strength publisher fails, a lower strength one is automatically switched in by the DDS middleware. In addition to fault tolerance, this can also help with load balancing on heavily used networks.

Switched fabric specifications already provide for a hot plug or hot swap capability. This hardware capability can be combined with a *virtual* hot plug capability at the application level using DDS middleware. Unlike traditional tightly coupled client/server architectures, DDS middleware allows producers and consumers to be dynamically added or removed in an operational system.

Many switched fabrics provide sophisticated features that allow, for instance, bandwidth-reserved, isochronous transactions across the fabric, something that is not supported by, say, Ethernet. Corresponding to the hardware QoS facilities, DDS-compliant middleware can offer a number of QoS policies that make predictability at the application level possible. For instance, the TRANSPORT_PRIORITY policy allows developers to manage how they prioritize one data flow over another.

The Road Map for Distributed Data Services

The existence of DDS as a standard specification endorsed by the Department of Defense (DoD) paves the way for addressing the challenge of distributing data among a myriad of defense systems. DDS is now mandated for data distribution by

The DDS Standard

The DDS for Real-Time Systems standard [3], from the Object Management Group (OMG), defines a publish-subscribe system that has high performance, is efficient, and offers a predictable way of meeting the data distribution requirements of data-critical systems with minimal overhead. The standard can be found on the OMG web site at <www.omg.org/technology/documents/formal/data_distribution.htm>.

The DDS standard has been in existence for almost two years, maps very naturally to the topologies and capabilities of switched fabrics, and is maturing into a solid technical approach to managing data distribution across large-scale distributed networks.

The DDS standard has three main goals:

1. To define a model for communication as pure data-centric exchanges, where applications publish (supply or stream) data which is then available to remote applications that are interested in it.
2. To provide a mechanism of specifying the available resources and providing policies that allow the middleware to align the resources to the most critical requirements, giving system designers the ability to control Quality of Service (QoS) properties that affect predictability, overhead, and resource utilization.
3. To permit systems to scale to hundreds or thousands of publishers and subscribers in a robust manner.

the Navy Open Architecture Computer Environment [4] and DoD Information Technology Standards Registry [5] and has already been adopted by programs such as Future Combat Systems, DD(X), Littoral Combat Ship, and Ship Self-Defense System.

But, despite the existence of a standard specification, the value of the solution is highly dependent upon its implementation. The specification defines certain features and capabilities, but not how they should be implemented.

A carefully designed middleware architecture can reduce the likelihood of a fault, limit the damage of a fault if it does occur, help detect faults immediately, protect the middleware from errors in application code, and isolate applications from errors in other applications. That architecture can also deliver significant advantages in the performance and flexibility of network distributed data communications.

For example, the DDS specification defines how a publish-subscribe communication model should work for a distributed real-time network. The DDS specification defines DataWriters for publishing and DataReaders for subscribing to a single topic on a user-defined data type. This in itself is standard and straightforward but how this is implemented can have a significant impact on network performance and scalability.

A robust implementation improves both performance and scalability by defining an architecture that supplies each DataWriter or DataReader with a queue that buffers messages bound for another endpoint through a transport. This architecture supports direct end-to-end messaging, since each endpoint (a DataReader or DataWriter) in each application directly communicates with a sister set of endpoints. Each endpoint has a dedicated set of buffers to hold messages in transit to other endpoints. This queuing architecture provides for an optimized transfer of messages from DataWriter to DataReader, no matter where each resides on the network. And because the endpoints queue and buffer transmissions to other endpoints, this architecture can easily scale to large and complex networks still with predictable delivery times.

In a similar manner, DDS defines the concept of a DomainParticipant, which is the fundamental container entity that can participate in a publish-subscribe network. A DomainParticipant can contain many DataReaders and DataWriters. Typical applications may use only one domain, and therefore have one DomainParticipant. However, applications are free to create several DomainParticipants so multiple instances of this entity can exist simultaneously.

Multiple execution threads are a way to optimize responsiveness and performance while also allowing the system to scale across a broad fabric-based network. One possible approach is to use several dedicated threads for each DomainParticipant in the following manner:

- **Event Thread.** DDS allows application designers to associate various QoS policies with each topic and data flow between a DataWriter and DataReader. These include timing related QoS that are implemented by the middleware. The Event thread manages both timing delays and periodic events such as protocol heartbeats, deadlines, and liveliness needed to meet the QoS policies requested by the application.
- **Database Cleanup Thread.** This thread purges old information from the internal data structures such as publication declarations and subscription requests.
- **Receive Threads.** A port represents a transport specific resource for receiving incoming messages. Data packets are delivered to transport's ports. Different DataReaders can be configured to receive messages on different ports. In order to minimize the end-to-end latency, a receive thread is created per port provided by the transport.

When the application writes new data to a topic, the message passes all the way through the middleware down to transport level send in the caller's thread. In the user's thread context, the message is serialized, deposited into the writer queue, encapsulated into a wire-protocol packet, and passed to the transport for delivery.

In the common case, the entire operation's critical path takes no inter-application locks and suffers no context switches. The event thread is only involved if the initial transport operation fails, or to execute follow-on processing such as ensuring reliable delivery. The event thread has ready access to the message since it is already stored in the writer queue. When the transport receives a new packet, the appropriate receive thread processes the packet, retrieves the message, stores it in the reader queue, and immediately executes the listener callback. In the common-case critical path, there are no inter-application locks or context switches. If the application requires the message to be handled with user threads, it can do so with DDS WaitSets. Both flexibility and performance are optimized, even as the network scales.

Performance can also be impacted through the poor use of the code execution path. Since lock contention can have a significant detrimental impact on performance, fast path optimization takes data to or from the network transport to the application using a single lock per message, greatly simplifying the resource sharing protocol.

Finally, instead of using lists to store the information needed to dispatch and manipulate messages, hash tables can be used. Although hash tables are more complex than lists, they have constant time access provided that the initial allocation of space is sufficient. Regardless, in the worst case, access time is logarithmic, which is better than linear linked lists.

The Implementation Optimizes Performance, Flexibility, and Reliability

Performance, flexibility, and reliability represent just a few ways that an implementation of the DDS specification can impact the three critical characteristics of data communication over a distributed network – reliability, performance, and flexibility. Alternatively, a poor implementation of the DDS specification can mean that the architecture works well under certain optimal implementations, but fails to take advantage of greater resources, and fails to scale as the network grows.

Data communications system developers do not want to change their application code when the fabric is updated, changed, or augmented. However, many possible implementations can deliver suboptimal results when the network topology changes. A DDS implementation can take this into account so that the application can be easily re-optimized to deliver a comparable level of performance in the face of evolving and changing fabrics.

As switched fabric technology advances, the middleware must support those advances by being able to adapt to new transport mechanisms and different resource requirements and availability. Being able to plug-in different transports in the middleware layer makes it possible to more easily incorporate new fabric technologies as they become available without making any changes at the application layer.

A superior implementation of the DDS standard enables network performance to be optimized to the particular application. It matches the performance needs with the underlying fabrics and availability of system resources such as memory. The design's flexibility allows it to target a broad array of applications and network topologies by supporting many transports and maintaining individual resources for each connection. Finally, the design avoids most key single points of failure, increasing reliability.◆

References

1. Arshad, Nauman, Stewart Dewar, and Ian Stalker. "Serial Switched Fabrics Enable New Military System Architectures." COTS Journal Dec. 2005 <www.cotsjournalonline.com/home/article.php?id=100438>.
2. Cotton, David B. "Switched Fabrics and the Military Market." COTS Journal. (Apr. 2005) <www.cotsjournalonline.com/home/article.php?id=100294>.
3. OMG. "Data Distribution Service for Real-Time Systems, v1.1." Document formal/2005-12-04. (Dec. 2005) <www.omg.org/cgi-bin/doc?formal/05-12 -04>.
4. Dahlgren Laboratory. Navy Open Architecture Computing Environment <www.nswc.navy.mil/wwwDL/B/OACE/>.
5. Defense Systems Information Agency. DoD Information Technology Standards Registry <https://disronline.disa.mil/>.

About the Author

Rajive Joshi, Ph.D., is a principal software engineer at Real-Time Innovations, Inc., where he specializes in the design of distributed and real-time systems. He has served as a consultant and developer for distributed systems projects in the areas of robotics and automation, including Alstom Schilling's Quest's remote-operated vehicle and CrouseHinds' automated filament-alignment system. Joshi is the author of *Multisensor Fusion: A Minimal Representation Framework*, and is a member of the Institute of Electrical and Electronics Engineers, Association for Computing Machinery, and American Institute of Aeronautics and Astronautics, Inc. He holds a doctorate and a Master of Science degree in computer and systems engineering from Rensselaer Polytechnic Institute, and has a Bachelor of Technology in electrical engineering from the Indian Institute of Technology in Kanpur, India.

Real-Time Innovations, Inc.
3975 Freedom CIR 6th FL
Santa Clara, CA 95054
Phone: (408) 200-4700 ext. 4754
Fax: (408) 200-4702
E-mail: rajive.joshi@rti.com

Letter to the Editor

Dear CrossTalk Editor,

As always, I thoroughly enjoy CrossTalk articles and most everything that comes out of the Software Engineering Institute.

I am of the considered opinion, after more than 42 years of development experience, that the problems with software quality can be attributed to a single cause, that being the inability to recognize complexity and act accordingly.

Resulting in: Do we tackle problems beyond our capability to solve using human intellect, excluding mathematical processes, even at the module level (unknown high levels of McCabe's Cyclomatic Complexity Index)?

Does anyone other than the Cleanroom Software Engineering sequence enumeration requirements analysis process folks identify and count the number of state transitions in a single module, much less a whole process or system? Or understand the implication of having 64 bimodal variables that can occur in any combination and various legal sequences? Or understand that programming is the mapping of state transitions to code?

Does anyone understand that the implication of Brooks' Law is a loss of intellectual control over the process and product as the process and staff and product grows in size and complexity?

Is complexity a self-inflicted wound? For example, could the failed IRS project been designed into multiple cases? Could a separate system have been created to process the majority of filers (e.g. 1040EZ), rather than a monolithic, all cases design? I expect so and would have been a quick success, though I recognize we may still be developing the more complex cases when Gabriel blows his horn. Yoder calls such monolithic design A Big Ball of Mud <www.laputan.org/mud/>.

Do programmers exacerbate the problem by not being able to defend their own or their team's performance, such as 12 defects per thousand lines of code for a medium complexity module?

Thanks for listening!

– Carl Wayne Hardeman
<cwhardeman@yahoo.com>

The Joint Services

STC

Systems & Software Technology Conference

18-21 June 2007 • TAMPA BAY, FLORIDA

Systems and Software Technology - Enabling the Global Mission

Presentation categories:

Rapid Response Capability
- Open Architecture
- Joint Rapid Acquisition Cell (JRAC)
- Disciplined Agile Development

Robust Engineering – Engineering for the Global Mission
- Systems of Systems Engineering
- Robust Software Engineering
- Software Product Lines
- Engineering for Manufacturing
- Adaptability

System Assurance – Addressing the Global Threat
- Information Assurance
- Software Assurance
- Anti-Tamper
- Open Source

Technology Futures
- New Computational Methods
- Time-Defined Delivery
- Technology Maturity

Communication Infrastructure
- Networks
- Interoperability
- Disaster Response

Enabling the Workforce
- Certification and Training
- National Security Personnel System (NSPS)

Don't miss this must attend event for:
- Acquisition Professionals
- Program/Project Managers
- Programmers
- System Developers
- Systems Engineers
- Process Engineers
- Quality and Test Engineers

Conference & Exhibit Registration Now Open - Join us in Florida by Registering Today!

TAMPA Bay

Complete schedule of presentations, summaries, speaker biographies and exhibitors can be accessed online at

www.sstc-online.org

(Un) Due Diligence

A few months ago, I wrote an article about transitioning to a new machine. In that article, I pointed out that I create frequent backups. And then backup the backups. And then – you get the idea. I indeed AM a bit obsessive compulsive about backups. In my office, I have lots of convenient backup options. I have network access to a RAID drive (Redundant Array of Inexpensive Disks, sometimes Redundant Array of Independent Disks). My RAID cluster gives me fault-tolerant storage of over a terabyte. By fault-tolerant, I mean that the RAID cluster consists of four disk drives, and any two of them can fail without losing any data. It's EXTREMELY reliable. Plus, we have it on a UPS. I also back up weekly to various USB thumb drives and two USB disks. I also create a hot backup on my laptop. Do I need so many backups of my backups? Not really. But when I DO need my backups, it pays off.

My fabulous new machine that I set up back in October failed a few weeks ago. Spectacularly. In fact, the magic smoke escaped from both the mother board AND the hard drive. (For those of you unfamiliar with the magic smoke concept, it's the mysterious substance that powers electronics. The magic smoke is held captive inside of a chip. If the chip ever breaks, the magic smoke leaks out, and the device will no longer work. This theory appears sound – in that whenever I see the magic smoke escaping, the device never works again.) Luckily, my total downtime due to yet another machine failure was about 20 minutes. Find unused machine, login to network, connect to RAID drive, access all files.

Does this mean that I will slack off on the excessive USB additional backups? Probably not. It has been said (Mark Twain) that if a cat accidentally sits on a hot stove once, it will never sit on a hot stove again. In fact, it will probably never sit on a cold stove again either. I have been burned on poor backups before – and lost a lot of work that took weeks to recover. And the problem with being burned is that you're scared of fire for a long time. It helps me sleep well at night knowing that there is redundancy in my backup process.

Back during World War II, the story goes that a B-17 returned from a bombing mission and was severely damaged. The Colonel had a meeting with his staff to discuss how the damage analysis could help them protect B-17s better. Looking at the damaged wings, everybody commented on how shot-up the airplane was and how additional plating was needed on the wings. Everywhere there was damage; somebody recommended additional modification to help strengthen the aircraft. After all of the staff had spoken, one lone lowly Lieutenant spoke up, and said, "This aircraft represents one of the bombers that made it back. What we should do is strengthen it wherever it is NOT damaged – because the damage we see is obviously survivable." Now THAT is thinking outside of the box.

That's the problem with processes – sometimes you are probably strengthening them where they have failed in the past. However, having been burnt, you might be ignoring weak points. What you need are processes that are adaptable and provide you with feedback. What kind of feedback? First of all, you need to know what your current status is. How do you get this status? Well, if you are a small development effort, you need to TALK. And I do not mean weekly status reports or End-Of-Month reports. I mean honest, open, frequent communication. This is the basis for ANY good process. Capability Maturity Model, Personal Software Process, Team Software Process, Scrum, Agile; call it what you want.

I personally like the concept of a daily stand-up meeting – where you literally stand up for the entire meeting. REALLY cuts out the long-winded talkers. In fact, my personal favorite method is when the person talking stands on one foot (not balancing the whole time – they can switch feet – but one foot should always be off the ground). In 10-15 minutes, issues get discussed, problems identified, resources quickly reallocate. While not every issue is resolved, at least proactive planning can occur.

What if you are too big for daily stand-up meetings? Well, you can still emphasize daily meetings for the programming teams, but somebody has to wrap up the important issues and elevate them. That is where metrics come in. Metrics are like the temperature of a project. You want to know whenever the project starts to run a fever. What do you measure? Whatever you need to measure that will allow you to reduce the fever. Errors. Fix times. Testing time. Source of errors. Rework.

Don't run a project based on how you got burned last time. Yes, it is important not to make the same mistakes again. It is also important to think ahead, and try not to make any critical new mistakes, either.

Backup often, but don't become obsessive. Use due diligence, not undue diligence. Instead of making one small part of your process bullet-proof, how about strengthening all (or at least most) of the weak points. Collect meaningful and useful metrics, and use the metrics to find out what the weak points are, and reallocate resources as necessary.

And try not to get burnt.

— David A. Cook
Senior Research Scientist and
Principal Member of the Technical Staff
The AEgis Technologies Group, Inc.
dcook@aegistg.com

Can You BACKTALK?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CROSSTALK, we also accept articles for the BACKTALK column. BACKTALK articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BACKTALK article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BACKTALK article, visit our Web site at <www.stsc.hill.af.mil>.

Software Engineering

As leaders in the avionics software industry, SMXG has a proven track record of producing software On-Time, On-Budget, and Defect Free.

Our staff of engineering professionals, as well as our industry partners, are committed to providing our customers with software and engineering solutions that make our Warfighters the most dominant forces in the world.

(a few aircraft we support)

Areas of Expertise:

- Navigation
- Radar
- Control and Displays
- Simulation and Emulation
- Mission Planning
- Defense Management System
- Prototype Development
- Electronic Warfare
- Operational Flight Software
- Weapon Delivery and Integration
- Avionics & Systems Engineering
- Network Centric Operations
- Turn-Key Test Systems and Solutions

- Test Program Set (Turn-Key)
- Interface Test Adapter (ITA) Design and Manufacturing
- Automated Jet Engine Testing (Hardware and Software Development)
- Engine Trending and Diagnostics
- Developers of Pacer Comet IV
- Software Control Center (SCC)
- Industrial Automation, Robotics
- USAF, Navy, NATO, & FMS customers

...PLUS MUCH MORE

Oklahoma City Air Logistics Center
76th Software Mantenance Group
Kevin D. Stamey (Director)
(405) 736.3340
DSN: 336.3340
kevin.stamey@tinker.af.mil

CROSSTALK / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CROSSTALK is co-sponsored by the following organizations: